ECE 655 Final Project Report

Luke Andresen, George Fang, Gavin Kitch

December 8th, 2024

Table of Contents

Introduction and Motivation:

For our ECE 655 final project, we set out to build a product that combined the embedded development with the internet integration techniques learned in the class to create a true IoT device. As avid basketball fans, we enjoy shooting around on our mini-basketball hoop at home. However, playing with small rubber balls usually leads to a tedious shooting session, where most of the time is spent chasing your ball around the room as it ricochets under couches and behind furniture. Being competitive people, we also wanted to track and compare our shooting statistics and were dissatisfied with the lack of existing solutions. Therefore, we saw the opportunity to employ our embedded development experience to create a device that returned the ball to the shooter and couple it with our database experience to allow for real time shot tracking. For the remainder of this report, this device will be referred to as SWISH (Smart Wired Interactive Shooting Helper).

System Design:

Based on our earliest brainstorming, we knew we could break the project down into five discrete parts: collection, detection/sorting, distribution, the database, and the web-app.

Collection:

For this project to work at all, we needed a reliable way to catch the ball, so that it could then be returned to the user. We also needed a way to determine if the shot was made for accurate stat tracking. The collection system thus had two core requirements:

1. Ball catching: This feature needed to catch **85% of balls that arrive within a 2 ft radius of the hoop** for us to consider it a success. It also had to be lightweight for wall mounting and had to funnel the balls consistently for later processing.
2. Make detection:This feature needed to detect **a ball passing through the hoop 95% of the time** for us to consider it a success. Our solution needed to be highly reliable in varied lighting conditions, fast enough to capture real-time events, and minimally invasive to the hoop's structure.

Detection:

The second key subsystem was a reliable processing mechanism (what we called the hopper). This entailed detecting when a ball lands in the hopper, identifying the user in the room, determining which user shot the ball, and sorting the ball to either a storage basket or a launching mechanism based on user presence. We decided that we wanted each user to have a specific ball so that the SWISH could easily pass each ball back to their owner as they shot it. The collection system thus had four core requirements:

1. Ball detection: This feature had to detect **a ball dropping into the hopper 95% of the time** for us to consider it a success. This detection system did not have the constraints of

size and real-time event accuracy though as it was not mounted on the rim and not detecting extremely rapid objects passing by.

2. Ball-to-user mapping: This feature had to **discern which user the current ball belonged to 95% of the time** for us to consider it a success. It had to work in various lighting conditions with a non-deterministic orientation of the ball in the hopper.
3. User location: This feature had to **locate the right user that corresponded to the current ball at least 75% of the time** for us to consider it a success. It needed to be able to locate users at the maximum depth of the room (about 15 feet) and with a wide field of view from the hoop, and work in non-consistent lighting conditions.
4. Ball sorting: This feature had to **drop the ball into the storage box or shooter with at least 95% accuracy** for us to consider it a success.

Distribution:

For the final physical system, we needed a way to dispense the ball to the user. The two key required features were simply:

1. Aiming: This feature had to **pass the ball back within a catchable range at least 75% of the time** for us to consider it a success. Therefore, the mechanism had to be able to aim around the entire room.
2. Launching: This feature had to **reach the user at a reasonable speed at least 75% of the time** for us to consider it a success.

Database:

Effective data storage served as the fourth subsystem component. We decided on a few key parameters that needed to be stored in order for our end-to-end solution to work properly: the name of the player who took the shot, a boolean to represent a make / miss, the region from which the shot was taken, and a timestamp. To be successful, **our database needed to properly store 99% of shots** such that we can accurately capture a snapshot of a user's statistics in the web-app.

Web-app:

Our web-app—a platform in which users could see a visualization of all of their data and interact with it—was the final component of the system. The implementation we constructed offered 2 separate views:

1. Global Leaderboard: The global leaderboard is an amalgamation of every user's statistical summary. It allows users to compare their progress and percentages to others, filtering by shooting percentage, total shots taken, and alphabetically.
2. Profile View: The profile view offers a detailed look into a user's shooting statistics. They can see their percentages, shots taken and shots made, as well as a heat map, which is a visual representation of their shooting percentages by section of the court, so they know their best shots, as well as what to practice more.

System Implementation:

Overall, we chose to build our system on a Raspberry Pi 4, as it provided a great balance of processing power, GPIO, web connectivity, and existing libraries and documentation.

Collection:

1. Ball catching: Based on existing similar solutions like full sized pass-rebounders, we immediately zeroed-in on a design consisting of a large net around the hoop.

Figure 1: Existing Rebounding Machine

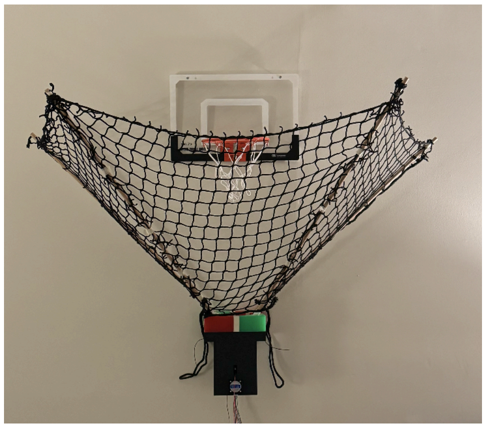


Figure 2: Our Collection Mechanism

Our design consisted of a 3D printed PLA base with 4 holes (2 facing parallel to the wall in opposite directions; 2 facing 45° away from the wall), all extending in opposite directions to house our wooden dowel stakes, and maximize the collection area. We used black polyester sports netting to wrap the wooden dowels creating an enclosure for balls to trickle into the main hopper.

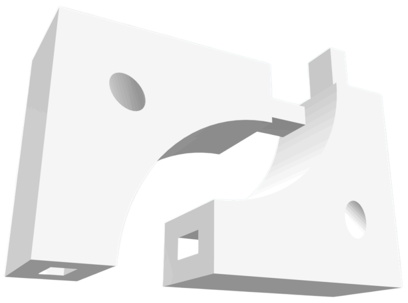


Figure 3: Initial Collection Holder Design

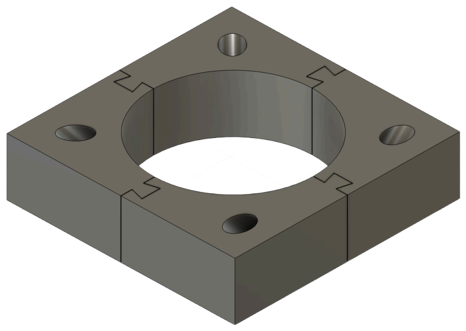


Figure 4: Final Collection Holder Design

Due to 3D printer space limitations, we had to split the base up into 4 parts. Our initial design used inserts to hold the components together as seen by the image on the left, while our final design took advantage of dovetail joints which led to a stronger and more reliable fit.

2. Make detection: To detect the ball passing through the hoop, we settled on using an IR line break sensor due to its small form factor and discrete digital output. An ultrasonic sensor's continuous output would have made real-time interrupts more difficult to implement.

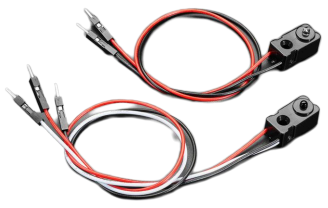


Figure 5: Line Break Sensor

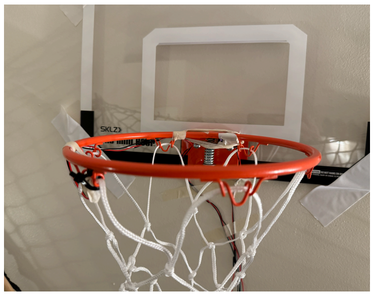


Figure 6: Line Break Sensor Mounted on Hoop

Detection:

With the ball now in the collection hopper, we executed the bulk of our detection operations all while the ball was stationary. In order to do this, we 3D printed a hopper that could house the majority of our detection components including a line break sensor, RGB color sensor, and stepper motor. Just off to the side as seen in figure 9 (due to Pi Camera cable length constraints), the Pi and wide-angle Pi camera were housed in a separate box.

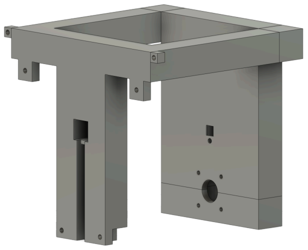


Figure 7: Hopper CAD

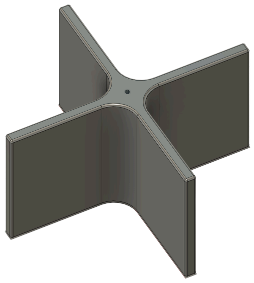


Figure 8: Pinwheel CAD

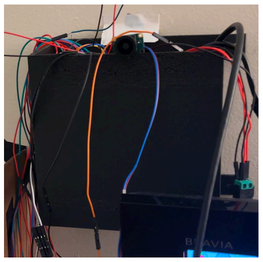


Figure 9: Camera Box

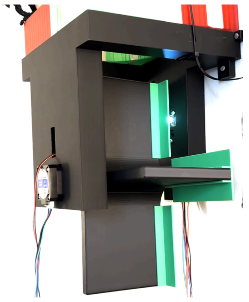
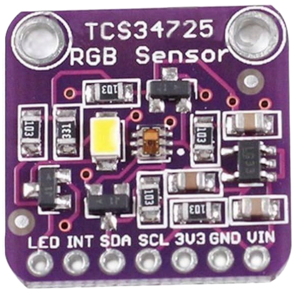


Figure 10: Full Implementation

1. *Ball detection*: For the same reasons detailed above, we used another line break to detect when the ball made it to the collection hopper. This line break was triggered regardless of a make or miss and notified the rest of the system that a shot was taken.
2. *Ball-to-user mapping*: The RGB color sensor then determined the color of the ball that was shot. To accurately track who took each shot, we assigned a unique ball color to each user in our system. Although we considered embedding RFID chips in the balls, we chose color detection as it avoids modifying



the balls, eliminates the need for additional sensors, and offers greater flexibility for future expansions.

Figure 11: RGB Sensor

3. *User location*: Now that the ball has been detected and mapped to a user, we settled on using facial recognition to find the user. As mentioned above, we purchased a high quality wide-angle camera to capture the full room and drove our system using OpenCV and the standard Python facial recognition library. As a starting point, we followed this tutorial and built our code upon the basic framework.

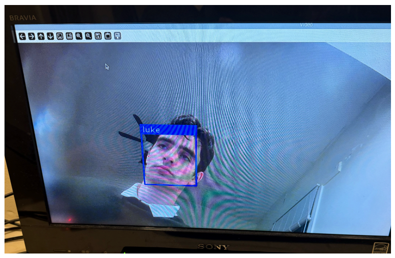


Figure 12: Working Facial Recognition

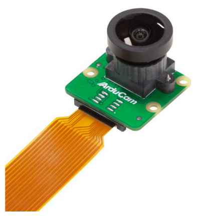


Figure 13: Wide Angle Camera

4. Ball sorting: Once the user was found, we elected to use a pinwheel design to spin the ball with the stepper motor for precise control of the angle, so that once the ball is dumped 90º to the left for storage or 90º right to the launcher, the system is back to its nominal state with an open gap for the next ball. For all 12V components, we maintained a 12V bus powered by a wall-plug AC/DC adapter. To control the NEMA-17 stepper, we used a DRV8825, where the Pi could control the direction and the step of the motor. We did run into some issues with the current draw of the stepper, but remedied it with careful tuning of the current limit on the driver.

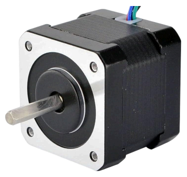


Figure 14: NEMA-17 Stepper Motor

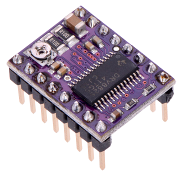


Figure 15: DRV8825

Distribution:

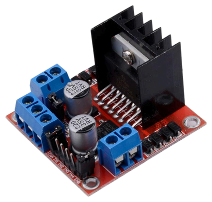


Figure 16: L298N

Figure 17: 5000 RPM 12V Motor

1. Launching: We knew that we would need DC motors to launch the ball, but we were unsure of the torque/speed ratio that would work best for launching the ball, and decided to start by testing with a 550 RPM motor. After testing, it became obvious that much higher RPM was needed, and little torque was required due to how light the balls are. Thus, we decided to use twin 5000 RPM motors driven by a L298N, where the Pi can control the direction of the motors using GPIO.

Once this was settled, we designed simple motor holders and a launch ramp that could hold the motors, plus flywheels to transfer the energy of the motors to the ball.

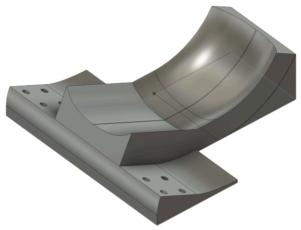


Figure 18: Ramp and Motor Mount

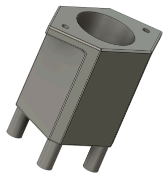


Figure 19: Motor Holder

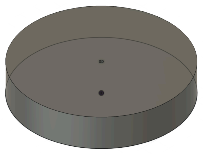


Figure 20: Flywheel

2. Aiming: To make the system easy to aim, we installed a wheel on the launching side to bear the load of the launcher. Initially, we built the system to be rotated with a stepper motor. However, we soon found that the large size of the launcher meant that the stepper did not have enough torque to move the device.

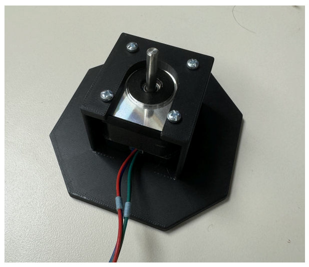


Figure 21: Original Stepper Design

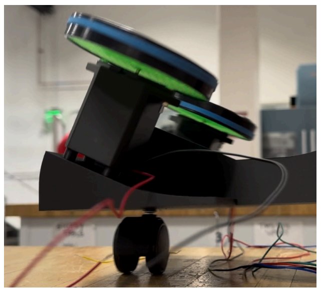


Figure 22: Wheel Installed on Launcher

Thus, we kept a similar design but installed a 150 kg servo in the steppers place, as it only needed 180º of rotation to cover the room.

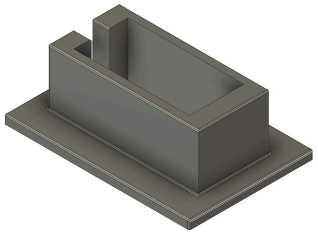


Figure 22: Replacement Servo Plate Design

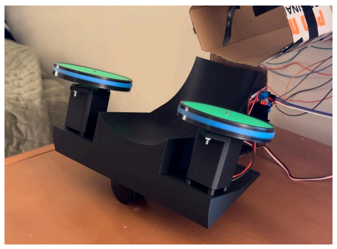


Figure 23: Completed Launcher

Embedded Development:

To handle ball detection, we made the line breaks trigger respective interrupts:

1. Rim line break: Toggles a made shot variable to true
2. Hopper line break: Executes pass back functionality

Once the hopper line break triggers the pass back functionality, the Pi first captures a frame from the camera to save where the user shot from. Next, the Pi uses GPIO to start up the launch wheels to give them time to reach max speed. The Pi then checks the image for faces and saves the data in global lists containing which faces were found and where they are in the image. Then, the color of the ball is sampled and mapped to a user using an internal table. If this user is not present in the global list of faces, then they must no longer be available and the Pi drives the stepper counter-clockwise 90º to drop the ball into the basket. Otherwise, the central position of the user's face is converted to a servo angle, and the servo is moved to that angle while the Pi drives the stepper clockwise 90º. While the ball rolls to the launcher, the system posts the user, the location (if the user was not found, the most previous location they shot from is assumed), and the made shot variable. Finally, the made shot variable is toggled to false. Then, after a brief delay, the motors power down and the system returns to its idle state. For more information, see the submitted code repository.

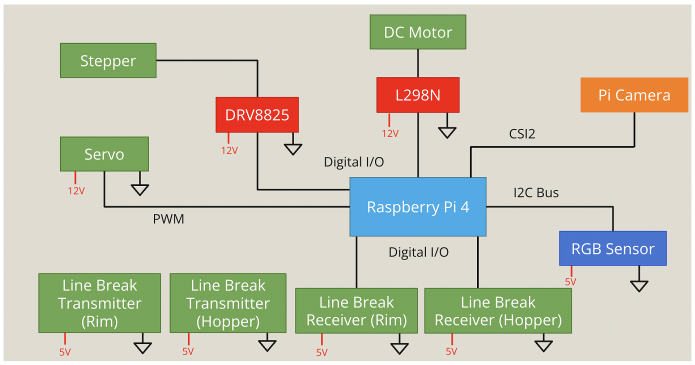


Figure 24: Final Hardware Schema

Database:

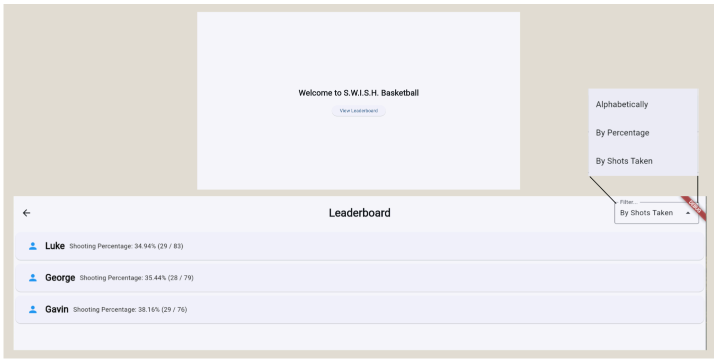
We hosted our PostgreSQL database in AWS on an EC2 instance. We handled HTTP POST requests for inserting collected by our device, as well as GET requests from our web-app for data visualization, with a Flask API. These processes are structured as follows:

1. POST request: The Raspberry Pi sent a post request for every shot attempt, wrapping all stats in JSON. This request was then handled by the '/add_shot' method, which constructed a SQL query to insert the delivered stats into the database with the aforementioned format.
2. GET request: To deliver JSON in the proper format, our GET request relied on a SQL query to condense all relevant entries in the database into a digestible format. This

method, '/all_user_stats', returned JSON for each user, listing their stats such that the web-app could display them.

Web-app:

The web-app interfaced with the database via an asynchronous 'fetchAllUserStats()' method, which parsed JSON via the Flask API and constructed a User (our custom model) for each entry. Stats for each user were then visualized based on the two views we deemed to be most useful for data representation. The leaderboard view (Figure 25), represents a combined statistical view of every user in the system. Using the filter option, users can be sorted alphabetically, by most shots taken, or by highest shooting percentage. Figure 26 shows the profile view, which visualizes a user's individual stats. The left column displays their basic leaderboard stats. The right column is a CustomPaint widget which overlays colors corresponding to their percentage values onto the court, so they can easily digest their shot efficiency by region.



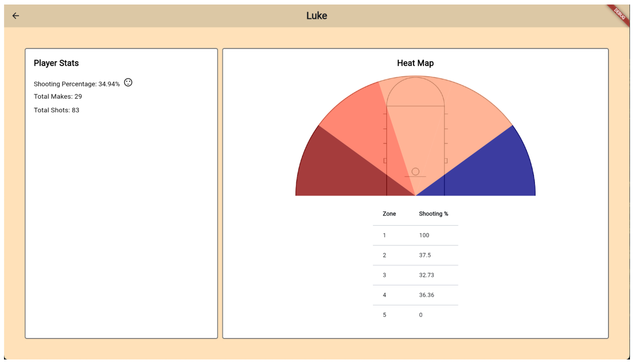
Leaderboard

Luke Shooting Percentage: 34.94% (29 / 83)

George Shooting Percentage: 35.44% (28 / 79)

Gavin Shooting Percentage: 38.16% (29 / 76)

Figure 25: Leaderboard View



Player Stats

Shooting Percentage: 34.94%

Total Makes: 29

Total Shots: 83

Zone	Shooting %
1	100
2	37.5
3	32.73
4	36.36
5	0

Figure 26: Profile View

Collection:

To test our collection mechanism, we took multiple shots from various angles to ensure the netting area was large enough to capture all reasonably accurate shots. We also tested the weight and durability of the system, ensuring it was light enough to be mounted on the wall using nails without causing significant sagging or deformation.

For the line break sensor, we wrote a simple script to output a value whenever the line was broken. Initially, the sensor only detected some of the balls passing through the hoop. Through testing, we identified two key sources of error:

1. *Alignment of the line break transmitter and receiver*: Accurate detection required near-perfect alignment of the transmitter and receiver. The spring-loaded mini-hoop caused hard rim shots to occasionally jostle the sensor out of place. After multiple iterations, we securely fastened both sides of the sensor to the rim to mitigate this issue.

2. *Bounce time parameter*: The line break sensor includes a "bounce_time" parameter, designed as a debouncing mechanism to prevent false readings. However, for our application, it introduced errors - when the shot was made and it was a perfect swish, the ball passed through the line break sensor faster than the “bounce_time” was set to. After testing, we determined that setting the bounce_time to 0 seconds provided the most accurate results. While this occasionally caused the sensor to detect multiple triggers for the same shot, it did not affect functionality. The global variable shot_made is simply set to true upon detection, and additional triggers have no further impact. Moreover, since the hopper line break sensor generates only one POST request per detected ball, disabling debouncing on the hoop sensor caused no issues.

Detection:

To test the remaining detection sensors, we ran similar iterative tests to ensure each individual component worked properly before integrating them in the final design.

1. Line Break:
    a. As mentioned above, the only difference between this line break and the one fixed to the hoop was the “bounce_time” variable which was set to 0.25 seconds to prevent any errant detections.
2. Stepper Motor:
    a. To test the stepper motor, we simply ensured we could spin it exactly 90 degrees to simulate depositing the ball to the right or left, and lining it up perfectly to accept a new ball. An issue with the stepper motor arose in relation to the line break sensor. When the stepper motor determined the direction to spin—either toward the collection basket or the shooting mechanism—we realized the line break sensor needed to remain continuously triggered until the ball was fully released from the pinwheel. To address this, we designed and attached covers to the sides of the pinwheel. These covers bridged the gap between the pinwheel arms and the ball, ensuring the line break sensor stayed engaged until the ball was deposited and the next empty chamber was correctly aligned.

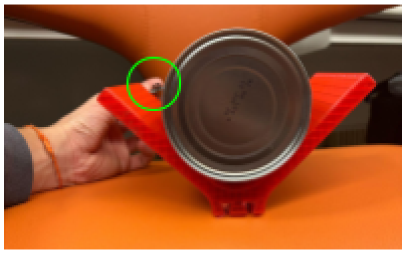


Figure 27: Pinwheel Without Covers

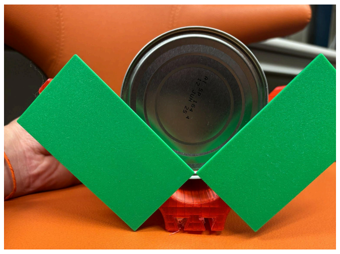


Figure 28: Pinwheel With Covers

3. RGB Color Sensor:
    a. The color sensor was fairly straightforward. As there were only three of us, we each chose one of three colored balls (red, green, blue), and the color of the ball

was determined by whichever of the three RGB metrics was the highest. For example an RGB of (21, 230, 76) would return green as the green integer is the highest. In the future, simple thresholding would allow for the addition of more balls

4. Camera/Facial Recognition:
    a. To accurately train the facial recognition model, we took multiple pictures of our faces from various angles and distances. A challenge we encountered was the variability in appearance due to changes in hairstyles, such as wearing our hair down, tying it up, or wearing a hat. To address this, we ensured consistency in our chosen hairstyle during both the training process and when interacting with the hoop.

Distribution:

As previously mentioned, we attempted to launch the balls at 550 RPM, but quickly found it wasn’t fast enough. We stepped up to 5000 RPM and found that it was the perfect speed once we added rubber bands for more grip and slightly increased the inflation of the balls for more compression.

Database:

In our testing of the system we took hundreds of shots, logging as each one was sent via POST request and validating that each entry had been saved via GET requests from the web-app. We determined that we indeed exceeded our 99% threshold for successful shots stored in the database, and were able to properly visualize our data in the variety of views on the web-app as well.

Discussions and Conclusions:

Our final product, SWISH, successfully met all the predefined metrics across its core components: collection, detection, distribution, database, and web-app integration. The system reliably caught and processed basketballs, accurately identified users and their shots, and seamlessly returned balls to their owners while logging shooting statistics in real time. Through iterative testing and refinements, we overcame challenges related to sensor alignment, facial recognition variability, and motor performance, ensuring a robust and functional IoT device.

This project not only demonstrated the practical application of embedded development and IoT integration but also highlighted the importance of interdisciplinary design, encompassing mechanical, software, and hardware engineering. Future work could focus on improving the scalability of the system, such as refining the color sensor thresholds for supporting more users, enhancing facial recognition for greater robustness in varied lighting, and optimizing the launcher for better precision for user depth. Ultimately, SWISH combines innovation and functionality, offering an interactive and data-driven solution for basketball enthusiasts while showcasing the potential of embedded IoT systems.