

Pong Fiction Final Report

1st Adonis Serrano
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
adonisse@mit.edu

2nd Sophia Torellas
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
storrell@mit.edu

3rd Daniel Reyes
Department of Mechanical Engineering
Massachusetts Institute of Technology
Cambridge, MA, USA
dreyes@mit.edu

Abstract—This project presents the preliminary design of a 2.5D pong game, named “Pong Fiction”. The Field Programmable Gate Array (FPGA) integrates data collected from an Inertial Measurement Unit (IMU) and a camera via signal processing, to create the game. The camera data dictates where the paddle appears on the screen in the x, y and z directions. The IMU acceleration data, which is communicated to the board through Bluetooth, is translated into a force exerted by a paddle onto the ball in the game. Once the data has been processed in the signal processing module, the game mechanics module calculates the position, size, and the overall physics and collisions of the ball. The graphics module renders the ball, paddle, and background for the game.

Keywords—Digital systems, Field programable gate array, Inertial measurement unit, Bluetooth, Computer networks

I. OVERALL DESIGN

The main goal of our project was to create an interactive game of pong in 2.5D space. Our game consists of a physical paddle and virtual paddle and virtual ball (to be referenced as “ball” henceforth). The virtual paddle is controlled by the physical paddle, and the ball is influenced by the collisions against the virtual paddle and walls of the game. The goal of the player is to prevent the ball from colliding with the imaginary sixth wall (which is equivalent to reaching its maximum size). We have successfully implemented this game along with a few stretch goals that will be outlined below.

To fulfill the 2.5D representation of pong, we have a physical paddle connected to an IMU to collect acceleration data, as well as a camera to get the location of said paddle. The data collected by the IMU is sent via Bluetooth and processed by the FPGA (Urbana model produced by AMD). The acceleration data is then displayed on the seven segment display located on the FPGA, and is used to drive the mechanics present in our game. Displayed on a monitor, we have a ball that grows, shrinks, and translates across the screen in response to hitting the virtual paddle or colliding with an in-game boundary.

Below we provide insight and schematics regarding the main components of our project, including the IMU to FPGA communication block, the processing of the data gathered from the IMU and the Camera, a description of the game mechanics and boundaries, a description of the seven segment display and the HDMI display.

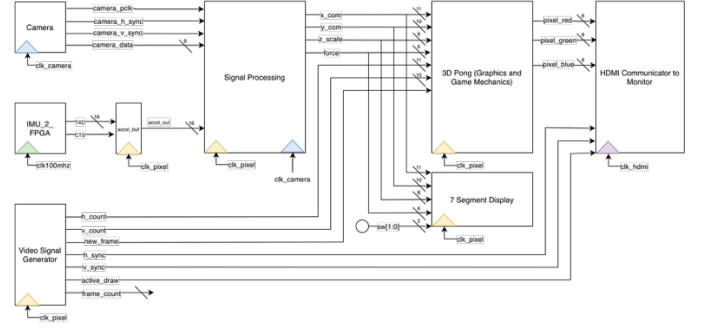


Fig. 1. High Level Schematic Diagram of 2.5D Pong Project

II. IMU_2_FPGA

This component oversees the collection of values from the IMU through a Microcontroller Unit (MCU) and sends the values to the FPGA. The IMU device used is an MPU-92/65, capable of collecting device acceleration (in the x, y and z directions) and orientation (about the calibrated x, y and z axis). The MCU being used is an ESP32 DevKitM-1, where it will communicate data with the IMU via I2C [1]. Once the data is collected with the MCU, we send data packets to the FPGA via Bluetooth. Both the FPGA and the MCU come with Bluetooth Low Energy (BLE) capabilities. The data being sent to the BLE chip on the FPGA is then transferred to the FPGA via UART communication for reconstruction and processing [1]. The figure below describes the high-level flow with how this module will function.

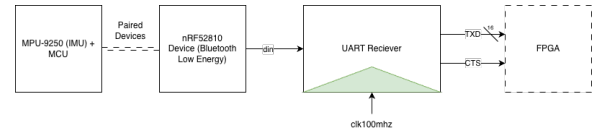


Fig. 2. Schematic of Bluetooth communication between IMU and FPGA

A. IMU Setup

To interface with the MPU9250, we rely on the MPU9250 Arduino library developed by Hideaki Tai [2], which provides a high-level API for configuring the sensor’s internal registers and accessing its accelerometer and gyroscope data. Using this library, we establish I2C communication between the microcontroller and the IMU by wiring the SDA and SCL pins to their corresponding ports and enabling the appropriate pull-up resistors. Once communication is verified, we run the library’s built-in calibration routines, which compute and store

the bias and scale factors for each sensing element. This calibration process corrects for manufacturing offsets and sensor drift, allowing the IMU to report more accurate acceleration measurements during data collection.

B. BLE Setup

With the use of BLE Arduino libraries, we set up the MCU to make Bluetooth connections. This process begins with a scan for the FPGA's BLE chip (identified by service UUID and target name). After the successful identification and connection, 6-byte packets of the x, y, and z acceleration are sent at a rate of 50 Hz. In the case of disconnection, we implement reconnection logic so we can continue to use the IMU [3].

C. Receipt and Reconstruction of Data Packets

The BLE chip on the FPGA board transmits incoming data packets to the FPGA via a UART interface. To support this communication, we updated the constraints file `top_level.xdc` to define the corresponding UART RX/TX pins and map them to the FPGA's I/O resources. The UART signal is then passed into a UART receiver module that samples the serial stream, detects the start bit, and reconstructs each byte according to the configured baud rate.

Since each acceleration measurement (x, y, and z) is represented as a 2-byte signed integer, we implemented an additional module to assemble the received bytes into complete sensor values. This module buffers incoming bytes from the UART receiver and, after six cycles, outputs the three reconstructed 2-byte acceleration values. The processed accelerometer readings are then made available to the rest of the FPGA design for downstream computation. The high-level of this module can be seen below.

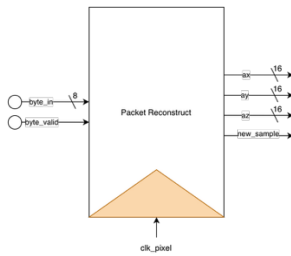


Fig. 3. Representation of Reconstruction of Data Packets

III. SIGNAL PROCESSING

The signal processing module takes in the peripheral data (camera data and IMU data) and outputs usable data used in the game (paddle x position, paddle y position, paddle z position, and paddle force on the ball).

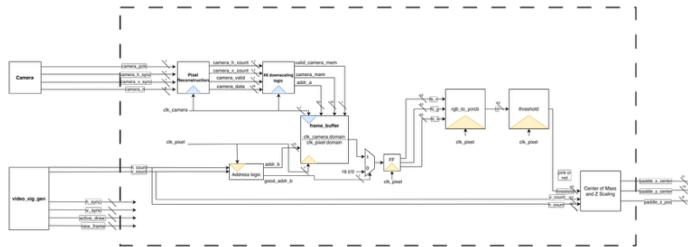


Fig. 4. Schematic of Camera Processing Module

A. Camera Processing

The camera connected to the board sends signals that are passed into the pixel reconstruction module. This generates the camera `h_count`, `v_count`, and `valid` signals as well as the camera data associated with a valid camera signal. A mux is placed after the pixel reconstruction to output when the camera pixel data is valid for use. The physical paddle we are using has a vibrant pink covering it, thus we turned the camera RGB values into the chroma red difference that allowed us to mask the output for chroma-red data between a specific threshold (in HEX: lower = 0x90, upper = 0xE0) picked up from our camera. We used the threshold value (1 if the chroma red difference is between the thresholds, 0 if not), in our center of mass (COM) and z-scaling module to calculate the x/y COM and our `z_scale`. The `z_scale` is slightly different from the COM calculations, as the amount of pink pixel points received from the camera determines the proximity of the virtual paddle to the screen.

B. IMU Processing

Due to the complexity of the orientation and tilt of the paddle, the z-direction of the acceleration is isolated and used to calculate a force value in the game. The force value is synthesized by the game mechanics module to produce the ball speed in the virtual setting. We decided that a greater acceleration value of the ball can be used to impact game difficulty, since a faster ball is harder to hit properly.

IV. GRAPHICS AND GAME MECHANICS

The graphics and mechanics module provides boundaries and rules for the ball and paddle to exist within the game.

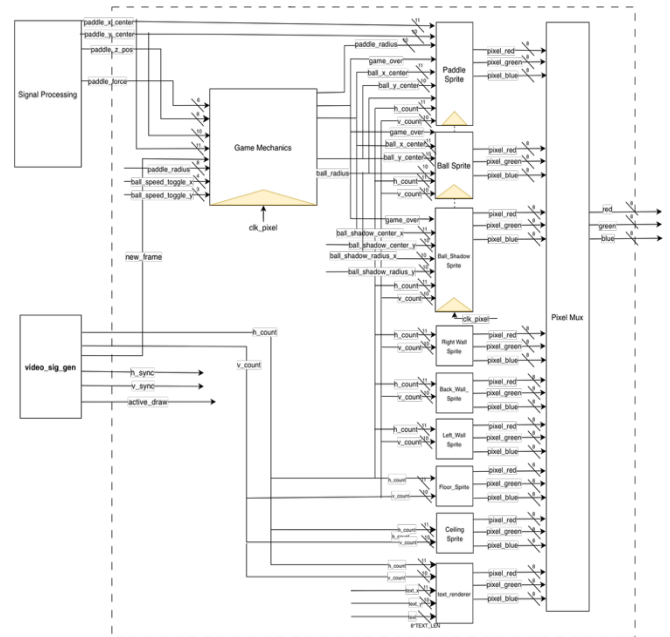


Fig. 5. Schematic of Graphics and Game Mechanics Module

A. Game Mechanics

The game mechanics module defines the rules of the game. It takes in the x-speed and y-speed of the ball, processed acceleration data from the IMU, and processed position data of the camera to determine the positions of the virtual ball and

virtual paddle. The module updates `ball_x` (center of the ball with regards to the x direction), `ball_y` (center of the ball with regards to the y direction), and `ball_z_pos` (z position of the ball), `ball_radius`, the ball collision boundaries, `game_score`, and `game_over` once per frame.

Before adding any new functionality in Verilog, we first tested it out in a Python simulation to ensure that the game works as expected. This has been useful for catching issues early and discovering solutions. Once the simulations met standards, a Verilog implementation was created.

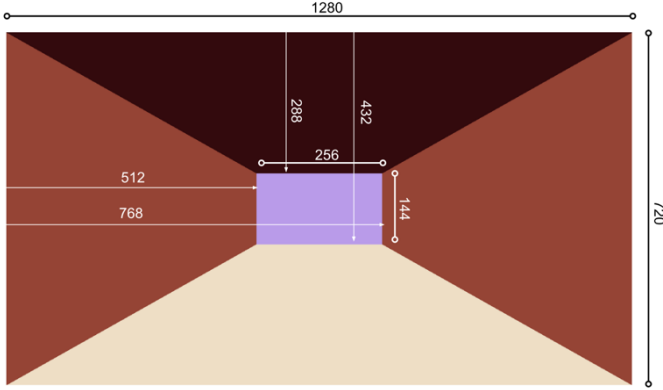


Fig. 6. Depiction of 2.5D Boundaries with Dimensions

All the graphics are displayed on a 1280 pixels x 720 pixels monitor. Our game is composed of five walls – four slanted walls depicting the side, bottom and top boundaries in perspective and one back wall depicting the z direction boundary. The purple back wall is 5 times smaller than the computer screen (the imaginary fourth wall). To maintain the illusion of 2.5D pong, the ball needs to grow and shrink. We decided to make the largest radius of the ball to be 128 pixels because it fit our boundary dimensions. In the 2.5D perspective, a smaller ball should be more restricted in space than a larger ball. The smaller ball is further away from the viewer, and thus we have to scale the boundaries with the ball. We knew that the smallest ball needed to be constricted to the purple back wall, so the ball would be restricted to 256 pixels x 144 pixels. The largest ball (128 pixels) would be able to move across the whole screen.

$$\frac{1280}{256} = \frac{720}{144} = \frac{128}{\text{smallest radius}} \quad (1)$$

To maintain perspective and according to (1), the smallest ball radius would be 5 pixels. The boundaries change based on how the ball scales. These boundaries are calculated and updated within `game_mechanics`. As part of the testing phase, we included white reference boundaries that grew and shrank with the ball in real time. These white lines can be switched on or off during the game. Following that the scale of the ball and boundaries are equivalent, a ball of radius 5 is bounded by 256 pixels x 144 pixels. The equations below describe equations denoting the position of the white reference boundaries.

$$\text{Leftmost Line} = 512 - [(\text{radius} - 5) \ll 2] \quad (2)$$

$$\text{Rightmost Line} = 768 - [(\text{radius} - 5) \ll 2] \quad (3)$$

Shifting by two is appropriate as dividing the maximum horizontal distance the ball must cover 512 pixels by the difference of the maximum radius and minimum radius (123 pixels) is roughly 4. We derived 768 from adding 256 to 512. Previously, we calculated the bottommost line and topmost line as a function of the background sprites (we wanted to ensure that the lines would intersect on the diagonal lines of the background). We have since changed this condition so that we can meet the timing requirement.

$$\text{Topmost Line} = 288 - [(\text{radius} - 5) \ll 1] \quad (4)$$

$$\text{Bottommost Line} = 432 + ((\text{radius} - 5) \ll 1) \quad (5)$$

Shifting by one is appropriate as dividing the maximum vertical distance the ball must cover (288 pixels) by the difference of the maximum radius and minimum radius (123 pixels) is roughly 2. For clarity, we derived 432 from adding 144 to 288. These lines defined the walls that the ball would collide against to allow for a 2.5D perspective. Due to our boundary approximations, even when the ball is the largest value it will still be bounded by the white boundaries, and not by the edges of the screen itself. If the ball ever collides with the boundaries, it bounces off and continues moving.

The ball can only collide with the paddle when the ball is within the same `z_position` as the paddle and when the ball and paddle overlap.

$$\text{ball_z} < \text{COLLISION_THRESHOLD} + \text{paddle_z} \quad (6)$$

`COLLISION_THRESHOLD` is a chosen value, which in this case it is defined as three. Below we define the equation that defines the time when the ball is overlapping the paddle.

$$(\text{b}_{\text{radius}} + \text{p}_{\text{radius}})^2 \geq (\text{b}_x - \text{p}_x)^2 + (\text{b}_y - \text{p}_y)^2 \quad (7)$$

When both conditions are fulfilled, the ball hits the paddle and goes into the back wall. Every time the ball hits the back wall, the `game_score` increments by `paddle_force`. We decided to increment the score by `paddle_force` because the faster the ball goes, the more uncontrollable it becomes. However, if the paddle doesn't meet the ball in time and the ball reaches maximum size, `game_over` is set to one and the game ends.

The ball is constantly moving in x, y, and z directions. The x and y directions are chosen at pseudorandom (we select a random bit from `paddle_x_center` and `paddle_y_center`).

B. Graphics

The background is composed of five different modules: the left sprite, ceiling sprite, right sprite, floor sprite, and back wall sprite.

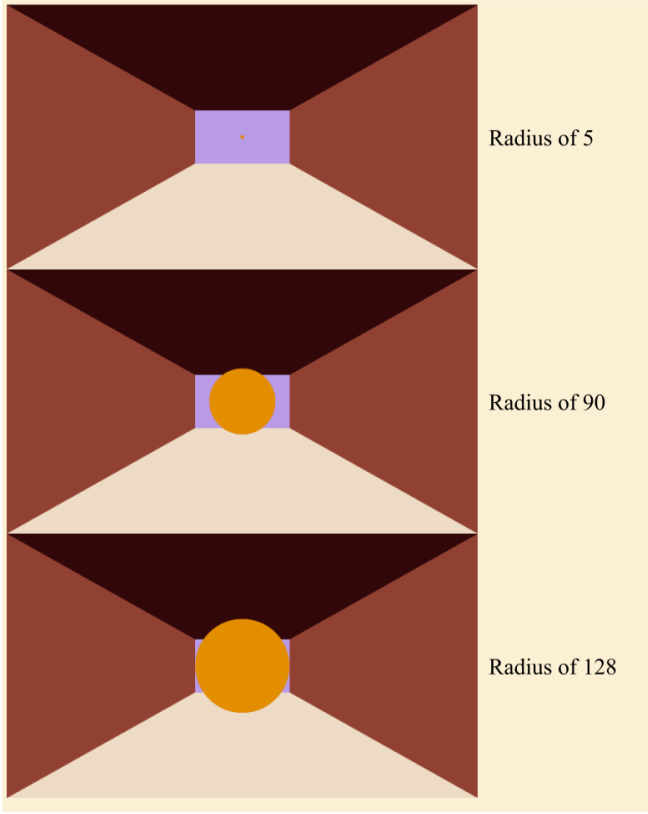


Fig. 7. Depiction of increasing ball radii and screen boundaries.

The first four sprites are defined by two lines, whose slopes and intercepts are displayed below.

$$y = \frac{9}{16}x \quad (8)$$

$$y = -\frac{9}{16}x + 720 \quad (9)$$

The pixels in the sprites are only colored if h_count and v_count fall within their respective boundaries. Otherwise, these values are set to 0. For our pixel mux, we choose the corresponding red, green, and blue values from each sprite using the logical operator “OR”.

The ball sprite is more computationally complex than the background. Initially, we thought that we wouldn’t be able to do all the multiplications for the circle in one clock cycle and had the `game_mechanics` module calculate the radius squared. However, we realized that with a bit of pipelining, we could easily calculate everything.

$$(x - b_x)^2 + (y - b_y)^2 \leq (b_{radius})^2 \quad (10)$$

We use [10] to draw the ball. b_x is the x center of the ball while b_y is the y center of the ball.

The paddle sprite is calculated similarly to the ball. The only major difference is that it takes as input the location of the ball and paddle to change the color of the ball in case it is being covered by the paddle. We had to pipeline this module in the same way we had pipelined the ball so that we could meet timing. Additionally, we draw the handle of the paddle to scale

with the size of the paddle. Nonetheless, the handle cannot hit the ball, so it is simply there for aesthetical purposes.

We also included a `text_renderer` so that we could display the score and `GAME_OVER`. We hardcoded lookup tables for each letter in `GAME_OVER`, `SCORE`, and numbers. The module takes text as input and finds the corresponding lookup tables for each letter. We output white pixels to write the words and score, which is then drawn onto the screen.



Fig. 8. Depiction of Score and Game Over sprites.

The last sprite that we added was the ball shadow. This was one of the stretch goals we successfully accomplished. Instead of a circle, we drew an ellipse that became more circular as it came closer to the screen. The x center of the shadow was the same as that of the ball (they needed to be in line with each other), but we had to ensure that the shadow would always project onto the beige floor. The bottom of the back wall is located at $v_count = 432$, and since the z_pos for the ball is between 0 and 255, we want the shadow to move between h_count values 432 and 687. Therefore, we ensured that the y center of the shadow was always $687 - ball_z_pos$. The x radius of the shadow is always the same as that of the ball, and the y radius is half of the ball’s radius.

$$c_1 = (b_{radius})^2 * (x - b_x)^2 \quad (11)$$

$$c_2 = \left(\frac{b_{radius}}{2}\right)^2 * [y - (687 - b_z)]^2 \quad (12)$$

$$c_1 + c_2 \leq (b_{radius})^2 * \left(\frac{b_{radius}}{2}\right)^2 \quad (13)$$

The equation of the shadow in relation to the ball variables is denoted by (13) and was referenced from an online source [4]. We calculate these values in the same manner we do for the ball and paddle sprites.

V. SEVEN SEGMENT DISPLAY

To see the values collected from the IMU or the values used for the game we decided to make use of the seven-segment display and the `sw[2:0]` to display each of the values. It displays data including the x,y, and z accelerations from the IMU, the x position, y position, and z positions of the paddle and ball, and the game state values (game score and if the game is over). Having the data displayed through the seven-segment system is useful for debugging.

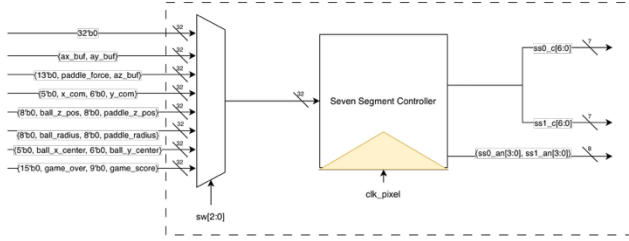


Fig. 9. Schematic of Seven Segment Display module.

VI. PROJECT ASSESSMENT

During the time of building out this project, we ran into a couple technical difficulties. When it came to module integration, we originally decided that everything would run off of `clk_pixel` (74.25 MHz). This conflicted with the timing needed to operate the logic used to set up the camera (200 MHz). To resolve this, we decided to run the camera at 200 MHz, and we included a single port BRAM buffer to resolve clock domain crossing. For game mechanics and some of the logic used to create our sprites, we utilize multiple multiplication and comparison operations between 22-bit and 44-bit numbers. Attempting to do this in one cycle made it difficult to meet timing requirements. To resolve this problem, we parallelized our logic to perform at most 1 multiplication per cycle and distributed our logic over at most 2 cycles. Since we made our calculations in game mechanics once every new frame, there was no need for pipelining because the calculations were made during the sync. When creating the module to reconstruct the IMU acceleration values, we ran into an error with byte alignment. Originally, `packet_reconstruct` would take in a sequence of any 6 bytes and spew out 3 16-bit values, looping through states that represent the receipt of each byte in a 6-byte sequence. Since we send the x,y, and z accelerations in 6-byte sized packets to the FPGA, there was a possibility that the alignment of each packet could've been off by a variable number of cycles. To resolve this, we added an additional byte that represented the start of a 6-byte sequence and added an additional state in our packet reconstructor that waits for this byte to be sent before attempting to join 6 bytes into the 3 16-bit acceleration values we need. Lastly, because of the large number of combinational computations made before sending the correct RGB values to the monitor, we needed to implement our `pixel_mux` using sequential logic rather than combinational logic. The changes described were all made and adjusted within our system to assure that timing is met.

At a minimum, we created all of the modules and sprites that would allow for the game to synthesize user input and function mechanically. These modules include the IMU to FPGA communication module, the camera and IMU data processing module, the game mechanics module, the boundary, ball and virtual paddle sprites, and a seven segment display module. We generate an on-screen paddle with x-y position and z scaling directed by a physical paddle with IMU and a camera. For debugging purposes, we also display the processed signal values from the camera and IMU on a 7-segment display. We also managed to complete a few of our stretch goals, including the evolving ball shadow and the perspective color changing for ball and paddle. We planned on using more of the IMU values to add

orientation and rotational complexity, randomizing the direction of the ball after collision with the paddle or walls, allowing for a change in player orientation, and making more modifications to the game graphics. However, due to the timeline and unforeseen obstacles, we have left these complexities for future implementation.

If we could restart this project, we would make a few adjustments. Logistically, we should have had more group work meetings, especially earlier on. This change would have reduced the challenge of integrating everyone's work together and would have made the project feel less daunting overall. We could also have had scheduled check-ins to hold everyone accountable and ensure the work was getting done at a good rate. In terms of technical modifications, we wouldn't change much, as we tweaked the project as we went, sometimes doing big overhauls to improve the code. However, one thing that we could have capitalized on more was using `h_sync` and `v_sync`. We calculated everything on `new_frame`, when we could have perhaps spread the costly calculations over a longer period of time. Another change we could have made was to reduce the size of some of the intermediate values. We ran into a lot of timing issues because of multiplying, and often there was no good way to reduce the number of bits as it was dependent on `h_count` and `v_count` (which were already big). Nonetheless, being more proactive in trying to minimize the size of the outputs would have substantially reduced the stress of meeting timing.

VII. SYSTEM EVALUATION

We wanted to make sure the system respected appropriate timing constraints and utilized low resources where necessary. We meet our set timing requirement with a positive slack of 0.569ns after placing and routing. We run the camera at 200 MHz, but the main speed of our system is 74.25 MHz (approximately 13.468 nanosecond clock period). With regards to throughout, our IMU sends 7-Byte sized packets to the FPGA at approximately 50 Hz, which our UART receives at a baud rate of 115200. We believe these specifications provide us with smooth graphics and great game playability. We made sure that Pong Fiction did not need much use for BRAM outside of resolving clock domain crossing, so we only used 32 blocks of BRAM. As for the DSP blocks we used, we utilized a total of 37 DSP blocks. The logic for our project requires a lot of additions and multiplications and many locations, so to try and cut down on cost we attempted to use shifts where needed and attempted to try alternative logic that would work similarly to multiplication or addition with less resources. Lastly, for timing requirements, we wanted to make sure that the game would function properly with a clock operating at 74.25 MHz. We managed to meet timing with a good amount of positive slack to spare. Overall, our final system was optimized for meeting timing and being low on resource utilization.

VIII. CONTRIBUTION

Overall, everyone worked together on the project. Adonis contributed to the setup of Bluetooth communication modules as well as setup for the MCU. Sophia and Daniel focused on game mechanics, sprite creation and camera processing. Daniel also focused on report organization while Sophia worked on

video editing. In summary, everyone contributed a piece to every portion of the project, but some portions were led by select team members

We would like to thank Hasan Zeki Yildiz for helping us brainstorm in the initial stage. Thank you to Monica Chan and Joe Steinmeyer for narrowing down our project to what it has become. Finally, we would like to thank Santiago Gonzalez for lending us his paddle for the project.

REFERENCES

- [1] Real Digital Staff, “Urbana Board Reference Manual,” [realdigital.org](https://www.realdigital.org/doc/496fed57c6b275735fe24c85de5718c2). <https://www.realdigital.org/doc/496fed57c6b275735fe24c85de5718c2> (accessed Nov. 26, 2025).
- [2] H. Tai, “MPU2950 Arduino library,” 2023, gitHub repository. [Online]. Available: <https://github.com/hideakitai/MPU9250>
- [3] Espressif, “Arduino ESP32: BLE,” 2025, gitHub repository. [Online]. Available: <https://github.com/espressif/arduino-esp32/blob/d1270e5/docs/en/api/ble.rst>
- [4] Wikimedia Foundation, “Ellipsoid,” [wikipedia.org](https://en.wikipedia.org/wiki/Ellipsoid). <https://en.wikipedia.org/wiki/Ellipsoid> (accessed Nov. 23, 2025).