ECE 4760 Lab 2 Digital Galton Board using RP2040

Ridhwan Ahmed and Nikita Dolgopolov

Lab Group 2

March 11th, 2025

High Level Overview:

The goal of Lab 2 is to develop a Digital Galton Board with the use of RP2040. The Galton Board is a device commonly used to showcase central limit theorem and gaussian distribution over many added random variables with identical distributions. Galton board consists of pyramid-shaped rows of pegs with bins underneath and balls dropping from the top of the pyramid and falling into bins. An additional goal of this lab was also to simulate as many balls falling through the Galton board as possible at a fixed FPS rate.

Some of the main features of the project include histogram and statistics display for the balls dropping down into the Galton board, sound production when balls hit pegs, and functionality to adjust number of the balls and their bounciness with the use of potentiometer.

Here is an example of what such a board might look like:

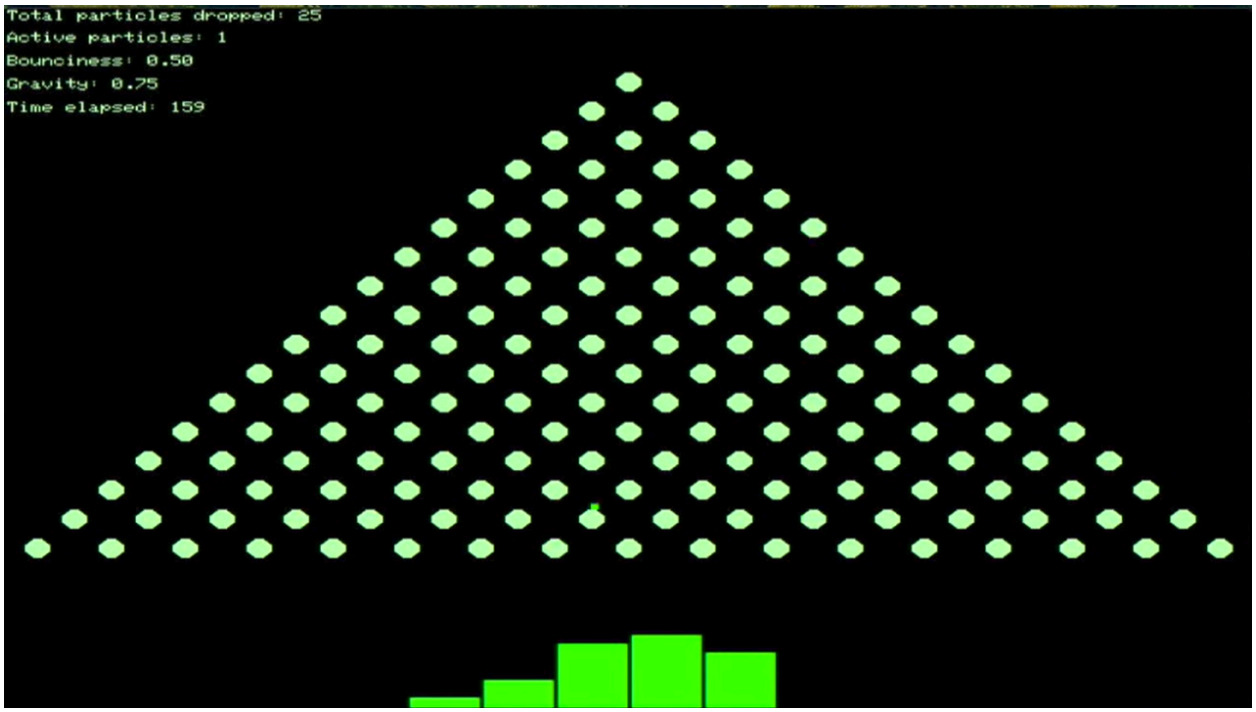

Figure 1: Digital Galton Board

The hardware in use for this project are RP2040 microcontroller, a reset button, a Digital-to-Analog Converter (DAC), a speaker, a potentiometer with a button, and a VGA screen with relevant circuitry and wiring.

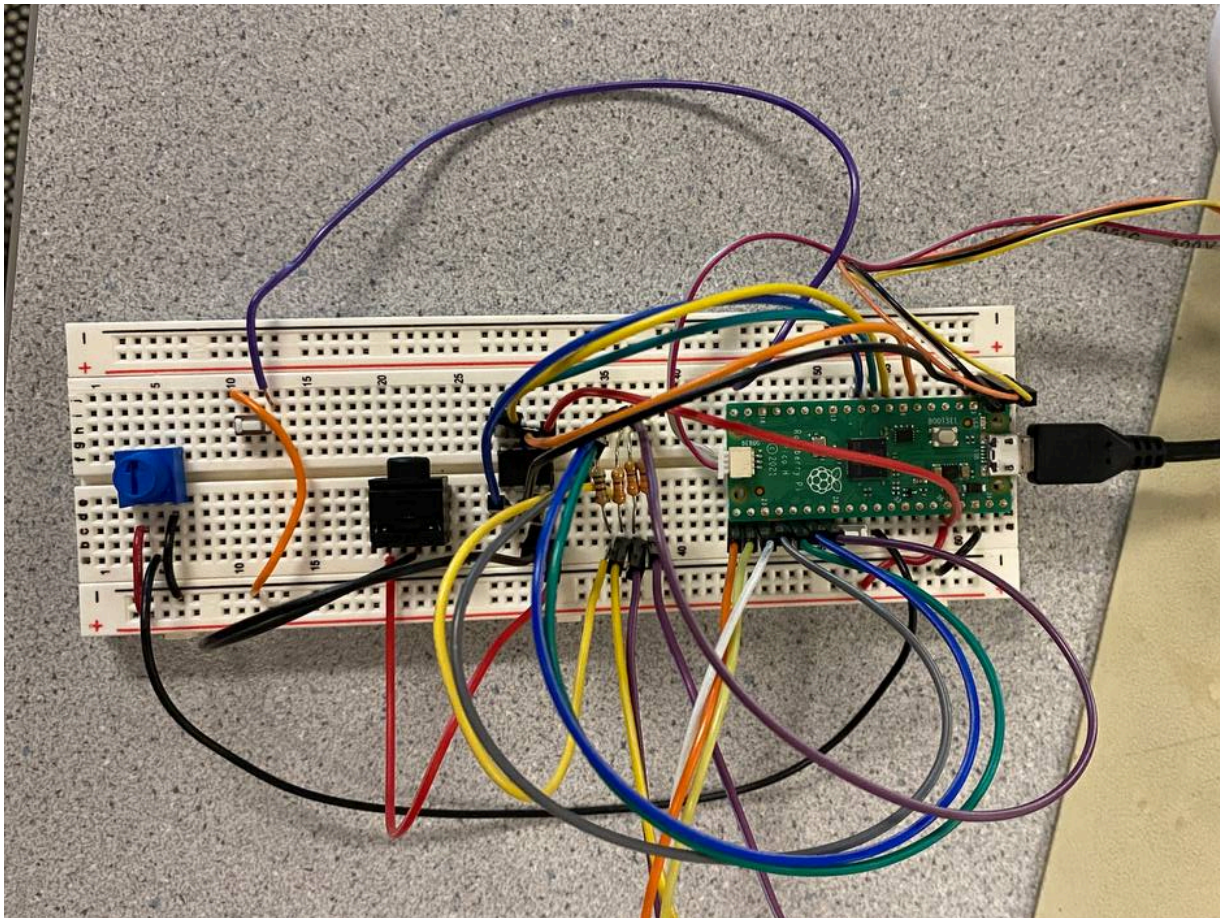

Figure 2: Actual Tested Circuit

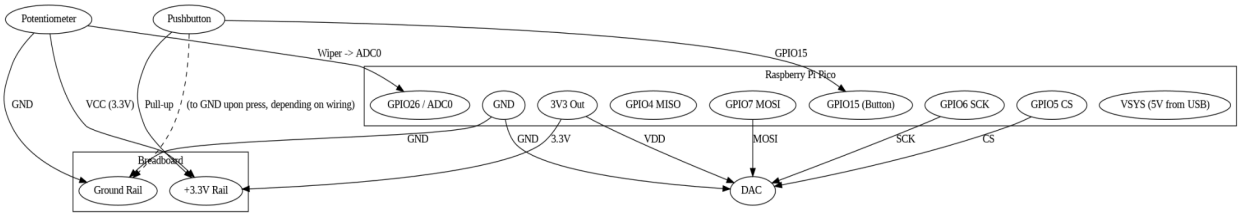

Figure 3: Mapped Connection Diagram for Test Circuit

The Software side of the project includes the following blocks: DAC communication via SPI, off-processor data processing and DMA channels, value look up in pregenerated table, states and semaphores, fixed point decimal arithmetics for speed, VGA screen communication and graphics, button press debouncing, and simulation of real-world physics to control the motion of the balls.

Software Logic Overview

High-level Overview

The flow of the program after initialization is as follows: check button and interpret its state with debouncing, calculate interactions of ball with the pegs and gravity and update their positions, update the histogram based on any balls falling into the bins, draw the new histogram and redraw any affected or dynamically changing objects, and display statistics. Additionally, when the ball strikes a new peg, a sound is produced.

For this, there are several graphics functions optimized to update only the critical parts of the screen, dynamics function that calculates the balls positions, object structs of balls and pegs to keep track of interactions, and helper functions to streamline calculations related to updating histogram, for example. For each of the software pieces, time efficiency was critical though we did not get to finish the project in its full ambition.

Objects

```
// -------------- Peg & Boid Structures --------------
typedef struct peg {
    fix15 peg_x ;
    fix15 peg_y ;
    int peg_row;
} peg;

struct peg *peg_array[150];
struct peg peg_obj_array[150];

typedef struct boid {
    fix15 x ;
    fix15 y ;
    fix15 vx ;
    fix15 vy ;
    peg *lastpeg ;
} boid;

struct boid *boid_array[1000];
struct boid boid_obj_array[1000];
```

Figure 4: Declaration of Structs

To define the balls(boids) and pegs, two structs were designed. Peg struct has fields that denote the physical x and y coordinates of the peg and the number of the row to which the peg belongs - this field allows simplifying some checks in the dynamics program to make it more time efficient. The boid struct holds the coordinates and velocities of each ball as well as a pointer to the last peg that the ball bounced off, which is used to control sound production when the ball hits a new peg.

The peg objects and the pointers to pegs are stored in two arrays sharing indices, which is done to simplify and accelerate some of the program logic. We rarely need to access the actual positions of each peg, but need to interact with their pointers pretty often, which are much easier to compare than two full

peg objects. Therefore, this decision allows for easy checking of whether the peg that ball interacts with is the same as previous time with the use of the function peg_compare.

Similar design decision is made for the boids. There are only 16 rows of pegs which amounts to less than 150 objects, which is the size of the peg arrays. The size of the ball array can be adjusted to any reasonably high number, which 1000 is a good initial estimate for. Because of these arrays, the objects are created during program initialization, so to change their properties or the number of objects we have, we need to access the objects at relevant indices and change the values in their fields.

For this, functions spawnBoids() and spawnBoid() reset the positions of the boids to the top middle of the screen, and the prior one also sets the randomized x velocity, sets the lastpeg field to NULL and makes sure to place this object inside of the boid objects array with pointer stored in boid array at the same index. The spawnArena() function iterates through the number of rows of pegs and number of columns in each row to spawn the pegs at their computed respective position by manipulating the fields of a new peg object by assigning x and y positions and row number to which the peg belongs. The object is then stored in the object array with the pointer stored in the peg array.

Dynamics: wallsAndEdges

To update the balls positions with time, the function wallsAndEdges() is called. It first iterates through the ball array and fetches the objects from it to access and modify their fields. For each ball, collisions with pegs need to be checked. To optimize this loop and avoid unnecessary collision checks, the row of the last peg that the ball collided with is checked; based on its value, only this and next 3 rows of pegs are checked, as the chances of the ball colliding with any other pegs on this iteration are almost zero. The pegs are fetched from the peg array only up to the number of pegs on the current board configuration, just like only the active number of balls is checked.

Checking for collisions technically requires calculating the square root distance between each peg and the ball; we do this only in the case that the x and y distances of the balls to the pegs are less than the sum of radii, which leaves some corner case yet stays a good approximation to the reality to save us computational time. After that, some physics is calculated to get the speed and direction in which the ball bounces off the peg and the sound is produced only if the collision happened with a peg different from the previous one. This is done by using the function peg_compare(first, second) that efficiently compares pointers to peg objects instead of looking at the values in the fields of the structs. The sound is produced by triggering a DMA channel.

After contact with the peg is evaluated, we check if the ball fell outside of the screen on the top, sides, or bottom. For the first two cases, the relevant velocity component is inverted to bring the particle back into few the next iteration. In the case of particle hitting bottom of the screen, several things happen: the x-position of particle is used to calculate the number of the histogram bin to which the particle is added, the histogram bin value is updated, the particle is respawned at the top of the screen with 0 y velocity and small random x velocity, and the counter of total particles dropped is incremented.

After all is done, the particle y-component of velocity is increased by the gravity value and x and y-positions of each ball are changed by the relevant velocity component values.

Additionally, some number operations are precomputed at initialization to save processor time and fixed point arithmetic is used to speed up all mathematical operations.

DMA and off-processor sound production

The audio output in this Galton Board relies on two chained DMA channels to stream a sine-wave table into the on-board DAC via the SPI at a fixed sampling rate. First, the code defines a sine table (**raw_sin** and **DAC_data**) of length 256, where each entry corresponds to a specific sample value for one cycle of a sine wave. When the system starts up, it fills this table by computing :

$$2047 \times sin(\theta) + 2047$$

For angles evenly distributed from 0 to 2π. Each raw sine value is then combined with a DAC configuration header (**DAC_config_chan_A**) to create a 16‑bit word in **DAC_data**. This approach ensures that each SPI transfer sends a properly formatted 16‑bit command to the external DAC channel.

The DMA setup begins with two claimed DMA channels: a “control channel” and a “data channel.” The control channel is configured to load the read address of the data channel with the base pointer to **DAC_data**. The data channel itself is configured to transfer 16‑bit samples from **DAC_data** into the SPI data register (spi_get_hw(SPI_PORT)->dr) in a continuous loop. A hardware “timer” (set to around 44 kHz) serves as the DREQ (data request) signal, causing the data channel to fetch the next sample on every timer tick. After the data channel processes all 256 samples in **DAC_data**, it automatically “chains back” to the control channel, which resets the data channel’s read pointer to the start of **DAC_data** again. This cyclical mechanism is what enables the DMA to continuously loop through the sine wave table.

Functionally, when the code calls **dma_trigger_beep()**, it starts the two DMA channels. The data channel fetches each 16‑bit entry from **DAC_data** at the DMA timer rate and writes it to the SPI output register, causing the DAC on the other side of the SPI bus to generate a steady sine wave. Once all 256 samples have been played, the code waits for the data channel to finish and then aborts both channels, ending the beep. This technique—chaining the control channel to the data channel and cycling through a prepared sine table—provides an elegant way to output continuous audio without tying up CPU resources. The CPU is free to run the Galton simulation and update the VGA display, while DMA autonomously handles the transfer of audio samples to the DAC.

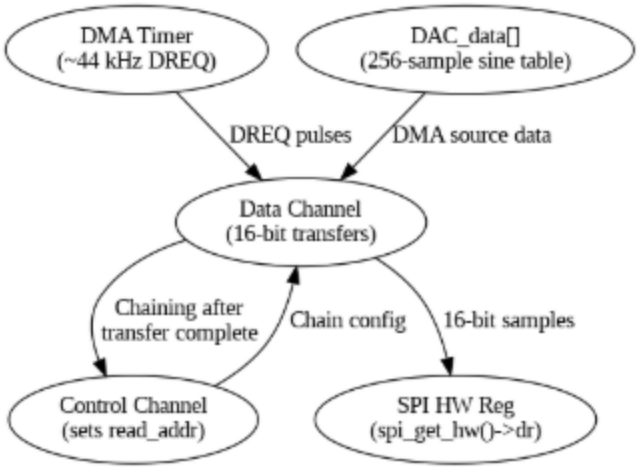

Figure 5: DMA functionality Diagram

Histogram

In our implementation the histogram is managed through an integer array, **histogram[],** whose indices correspond to exit slots at the bottom of the Galton board. Each time a ball reaches the bottom (detected by a "hit bottom" condition in the code), the ball's x-coordinate is mapped to a particular bin index using a pretty straight forward integer calculation. Specifically, the x-position in pixels is offset by an initial reference point and then divided by the horizontal spacing between bins. Once the correct bin index is determined, the corresponding element of the histogram[] is incremented by one to reflect that another ball has exited that slot.

In order to visualize these counts each frame, the code first scans through all the bins to identify the maximum bin count (stored in **maxcount**). This value is used to normalize each bar's height, ensuring that the bar with the highest count occupies the full vertical space. After clearing the previous frame's bars. After clearing the previous frame's bards by drawing black rectangles over their old positions, the program computes the new bar height as:

$$bar\_height = (histogram[i] \div maxcount) \times histo\_height.$$

Using this scaling factor, it draws each bar as a rectangle whose top edge is located at **histo_base_y - bar_height** and whose fixed width is **bin_width**. The bars are spaced horizontally according to the bin index, with the left-most bin at **bin0_pos** and each subsequent bin offset by **bin_width** pixels. Thus, as balls continue the drop, the code works to continuously update and redraw the histogram, which aims to produce a real time graphical distribution of how the balls exited the Galton board. Whenever the board is reset (for example, when the user changes the number of boids or the bounciness via the potentiometer), the function **reset_histogram()** is called to set all bin counts to zero and clear any prior distribution.

Graphics

There are several items that need to be drawn on the screen: the balls, the pegs, the histogram, and the statistics.

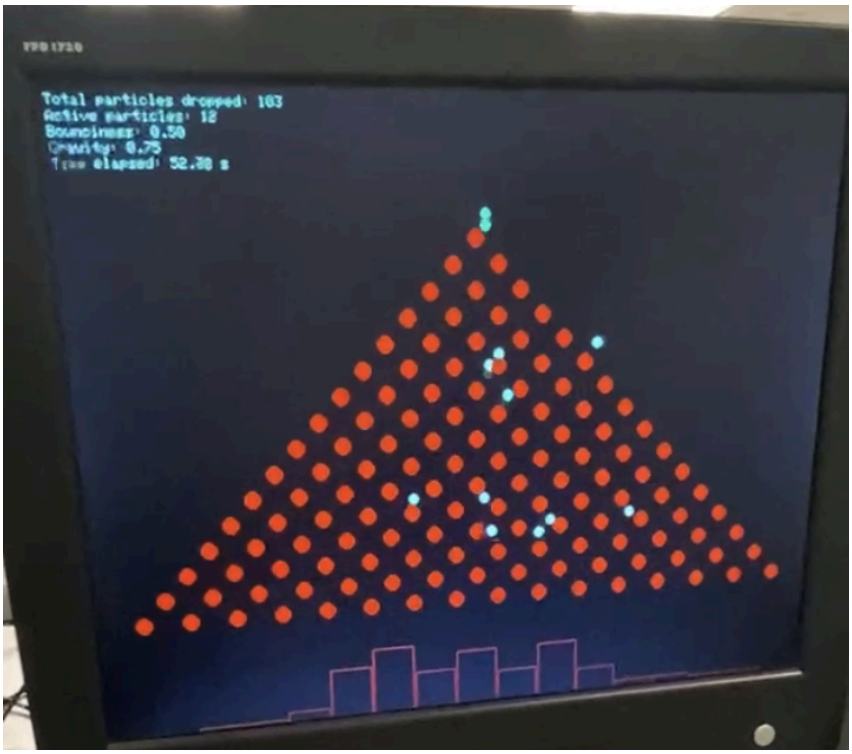

Figure 6: Digital Galton Board on VGA Screen

To redraw the balls, the logic is simple: when the ball is fetched from the array inside of the wallsAndEdges(), its spot on the screen is painted in black - the color of background. The ball is redrawn at its new updated position at the end of the loop with its original color. This is done for every ball.

There is a chance of a ball teleporting inside of a peg due to discretization of time - if it is too large and the ball position is not updated often enough, part of a peg would be colored in black, which was done to erase the old ball position. Therefore, pegs need to be repainted after all ball positions are recalculated and balls are painted. This is done with a drawArena() function that fetches the position of each peg in the peg array and draws the circles at it.

The histogram is drawn as a series of rectangles whose heights correspond to the amount of particles that had fallen in each of the bins and are located between consecutive pegs. The heights are scaled by a fraction of the number of balls in the bin to the maximum number of balls in all of the bins. In the code that we implemented, to redraw the rectangles, the whole histogram space was painted black and then new histogram with recalculated heights was drawn using unfilled rectangles to reduce # of pixels for redrawing. This approach can be improved by painting the frames of rectangles black instead of the whole space - this would cut the number of pixels to redraw by 1-2 orders of magnitude.

To display statistics, the text is broken down into two parts: static and dynamic. The static text includes the part of the message that stays unchanged throughout the program, such as the labels to the provided statistics (e.g. number of balls and bounciness), and is drawn at initialization. The dynamic part of the text, such as the actual values of parameters that can be changed and total number of particles dropped, actually has to be updated and is redrawn on every frame. It is a fair assumption that the static part of the text is not gonna be damaged by the balls flying around the screen, as it is in the top left corner - pretty far away from all the boards and the board. Therefore, we don't need to waste CPU cycles on redrawing it.

Potential improvements

The first improvement to be made is redrawing the histogram using wireframe rectangles instead of filling them with black when erasing, as outlined in the graphics section. Another potential improvement related to graphics is related to changing the way fillCircle function is implemented. This function is used to draw the pegs and the balls and it requires quite a bit of computing. Mapping the circles and drawing them pixel by pixel would be a more time effective though less elegant solution.

Inside of the dynamics loops, we could use an alpha max beta min algorithm to approximate the distances between centers of the pegs and balls, as it is a more time efficient way of getting an approximate square root of the sum of squares value than convert between fixed points arithmetics and floats and computing a square root of the float - the current implemented approach. We have started adding it and developed the helper functions, but did not get to debug it.

Additionally, using the second core on the RP2040 would allow drawing many more objects on the screen by, for example, computing all dynamics on that core and leaving graphics to the first one. This would require careful use of semaphores to ensure that the animation or dynamics is not being intervened by the other core, as it can cause many issues related to shared variables.

Potentiometer and Button Inputs

In this system, pushbutton input is handled by a combination of edge detection and debounce control. The button is wired with a pull‑up, so its digital input reads high when unpressed and low when pressed. Each frame, the code calls a **check_button()** function that samples the current pin state **(curr_val)** and compares it with the previous state **(last_button)**. It looks specifically for a falling edge—the transition from true (unpressed) to false (pressed). Because mechanical switches often produce rapid on/off transitions (this idea from “switch bounce”), the code implements a software debounce by requiring that at least a certain time interval (DEBOUNCE_DELAY_US, e.g. 20 ms) has elapsed since the last valid press (tracked by last_debounce_time_us). Once a valid falling edge is detected, the code updates a global UI state (**ui_state**) in a cyclical manner (e.g. 0 → 1 → 2 → back to 0). That state variable dictates which parameter the user is currently adjusting. For instance, in “state 1” the user controls the number of boids, while in “state 2” the user modifies the bounciness factor. Whenever a state change occurs, the code also resets key data structures (such as the histogram array) to reflect that a new parameter is now being tuned.

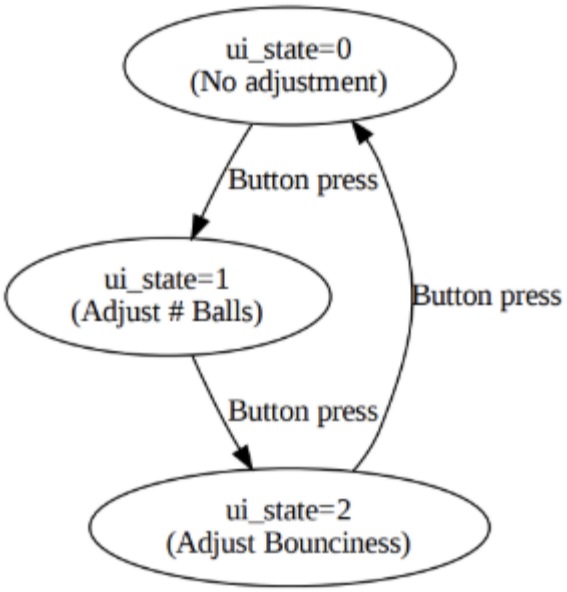

Figure 7: Button Press State Machine

Parallel to the button logic, the potentiometer is read via the Pico's ADC hardware each frame. Specifically, the code uses adc_read() to get a 12‑bit value and then shifts or scales it down to an 8‑bit range (0–255). Depending on which UI state is active, the system maps this 0–255 value into the relevant parameter. In the case of adjusting the boid count, the code scales the raw reading by max_boids / 256 and assigns the result to num_boids. If the user is in the bounciness‑adjustment state, the same raw ADC input gets interpreted as a floating‑point coefficient between 0.1 and 1.0, updating a global bounciness variable (bounciness = float2fix15(0.1 + 0.9 * (pot_val / 255.0f))). By separating the parameter‑selection logic (through the button) from the continuous control logic (through the potentiometer), the code provides a flexible, multi‑mode user interface. This design lets the user cycle the button through various adjustments, and then fine‑tune the selected parameter by turning the potentiometer—while the rest of the system (e.g. the Galton board simulation and histogram drawing) continues to run in real time.

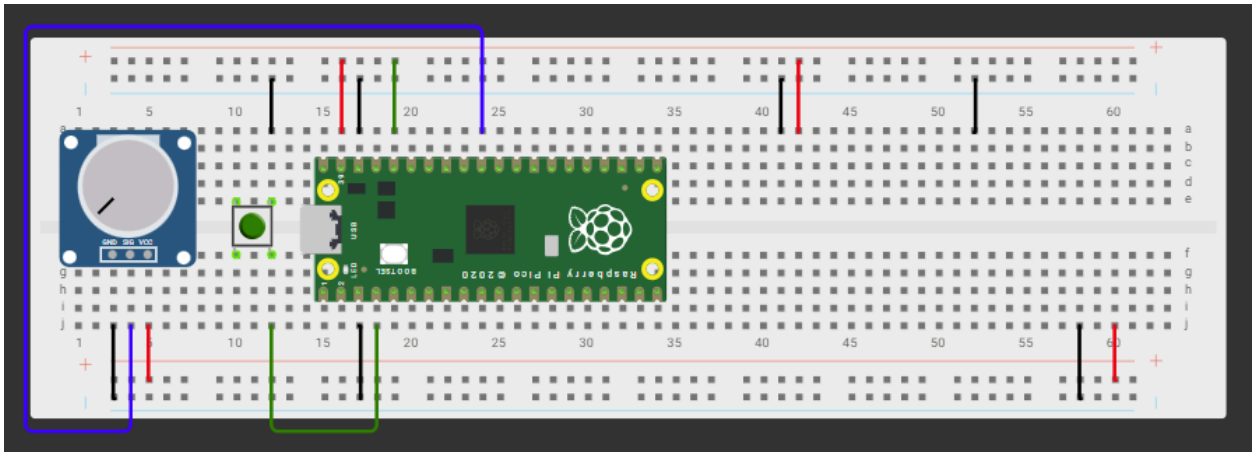

Figure 8: Micropython Architecture to understand potentiometer and button integration

Debugging and Testing Strategies

During development, we used several testing and debugging strategies to ensure the system behaved as expected. First, when calculating peg-to-ball collisions, we attempted to replace the standard square-root distance check with an alpha-beta max approximation in hopes of improving performance; however, this caused erratic collision responses and led us back to the true square-root method for accurate distance measurement. For the button state machine, we watched serial monitor output to verify that each button press advanced the system through the expected states (e.g., “adjust number of boids,” “adjust bounciness,” etc.) and correctly triggered a histogram reset. Next, we focused on histogram normalization, intentionally dropping multiple balls into a single bin to confirm that the tallest bin stretched to the full histogram height and that other bars scaled accordingly. Finally, each time we observed a button press leading to a state transition, we verified that the code properly cleared the histogram array so that subsequent distributions would be displayed from a fresh starting point.

When we attempted to scale the Galton board simulation from a few dozen balls up to 1,000 or 2,000, the system quickly became overwhelmed and dropped below the required 30 FPS. The main culprit was the collision-detection loop inside wallsAndEdges(), where each boid tests for potential impacts against every peg. As the number of balls grows, this produces a rapidly increasing number of trigonometric or square-root calculations per frame, creating a computational bottleneck. In an effort to relieve pressure on a single core, we explored a multicore proto threading technique, placing all collision physics on one thread while offloading histogram calculations and drawing onto another thread. We hoped that concurrent execution would smooth out frame times and allow more balls to run at real-time speed. Unfortunately, concurrency overhead—especially without carefully synchronized data sharing—still caused stalls and visual artifacts; even with the second core, the aggregate workload remained too large to sustain 30 FPS once we exceeded roughly 60 balls. Consequently, while distributing tasks across both cores reduced some stuttering, it ultimately proved insufficient for handling upward of a thousand boids, and further optimizations (e.g. spatial partitioning or a more approximate collision method) would be needed for such large-scale simulations.

As for some other general strategies, we ensure to isolate each change in the program by module testing the program: observing values of variables through print statements, LED blinks to check that certain operations are executed, and corner cases such as “single boid-single peg”. Additionally, when redrawing objects, we often replaced the initial clearing black color to green or blue, as they would be easily visible. This would allow us to confirm that the objects were erased correctly.

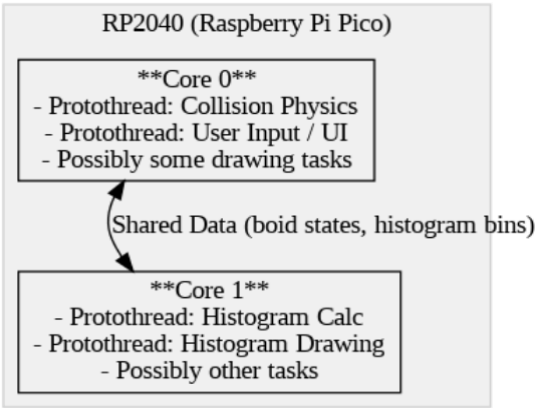

Figure 9: Protothread Design Architecture

Appendix:

```
/**
 * Hunter Adams (vha3@cornell.edu)
 * Modidfied by Nikita Dolgopolov and Ridhwan Ahmed
 *
 * This demonstration animates a Galton board on a VGA screen using the
RP2040.
 * Balls (boids) bounce down a grid of pegs. A histogram below shows the
distribution
 * of where the balls exit. Text on the left shows relevant parameters.
 *
 * HARDWARE CONNECTIONS
 *  - GPIO 16 ---> VGA Hsync
 *  - GPIO 17 ---> VGA Vsync
 *  - GPIO 18 ---> 470 ohm resistor ---> VGA Green
 *  - GPIO 19 ---> 330 ohm resistor ---> VGA Green
 *  - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
 *  - GPIO 21 ---> 330 ohm resistor ---> VGA Red
 *  - RP2040 GND ---> VGA GND
 *
 * RESOURCES USED
 *  - PIO state machines 0, 1, and 2 on PIO instance 0
 *  - DMA channels (2, by claim mechanism)
 *  - 153.6 kBytes of RAM (for pixel color data)
 *
 */

// Include the VGA grahics library
#include "vga16_graphics.h"

// Include standard libraries
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <string.h>

// Include Pico libraries
#include "pico/stdlib.h"
#include "hardware/dma.h"
#include "hardware/spi.h"
#include "pico/divider.h"
#include "pico/multicore.h"

#include "hardware/gpio.h"
#include "hardware/adc.h"

// Include hardware libraries
#include "hardware/pio.h"
#include "hardware/dma.h"
#include "hardware/clocks.h"
#include "hardware/pll.h"

// Include protothreads
#include "pt_cornell_rp2040_v1_3.h"

// === the fixed point macros ========================================
typedef signed int fix15 ;
#define multfix15(a,b) ((fix15)((((signed long long)(a))*((signed long
long)(b)))>>15))
#define float2fix15(a) ((fix15)((a)*32768.0)) // 2^15
#define fix2float15(a) ((float)(a)/32768.0)
#define absfix15(a) abs(a)
#define int2fix15(a) ((fix15)(a << 15))
#define fix2int15(a) ((int)(a >> 15))
#define char2fix15(a) (fix15)(((fix15)(a)) << 15)
#define divfix(a,b) (fix15)(div_s64s64( (((signed long long)(a)) << 15),
((signed long long)(b))))

// Wall detection
#define hitBottom(b) (b>int2fix15(380))
#define hitTop(b) (b<int2fix15(10))
#define hitLeft(a) (a<int2fix15(10))
#define hitRight(a) (a>int2fix15(630))
```

```
fix15 max(fix15 a, fix15 b){
    if (a>b) {
        return a;
    } else {
        return b;
    }
}

fix15 min(fix15 a, fix15 b){
    if (a<b) {
        return a;
    } else {
        return b;
    }
}

// uS per frame
#define FRAME_RATE 33000

// A color used for the boids
char color = WHITE ;

// -------------- Peg & Boid Structures --------------
typedef struct peg {
    fix15 peg_x ;
    fix15 peg_y ;
    int peg_row;
} peg;

struct peg *peg_array[150];
struct peg peg_obj_array[150];

typedef struct boid {
    fix15 x ;
    fix15 y ;
    fix15 vx ;
    fix15 vy ;
    peg *lastpeg ;
} boid;
```

```
struct boid *boid_array[1000];
struct boid boid_obj_array[1000];

// -------------- Physical parameters ---------------
fix15 bounciness = float2fix15(0.5);
fix15 gravity    = float2fix15(0.75);

fix15 boid_r = int2fix15(4);
fix15 peg_r  = int2fix15(6);

// -------------- Number of boids & rows -------------
int num_boids = 30;
int max_boids = 60;

// Changed num_rows to 16 for the full Galton board
int num_rows  = 16;
int num_pegs  = 0;

// -------------- For collision math ---------------
fix15 normal_x = 0;
fix15 normal_y = 0;
fix15 distance = 0;
fix15 dx = 0;
fix15 dy = 0;

// -------------- Potentiometer ---------------
// const float conversion_factor = 0;
uint16_t potentiometer = 0;
uint16_t potentiometer_last = 0;

// -------------- Timing & histogram ---------------
// Keep track of total balls that have exited
int total_dropped = 0;
int maxcount = 0;

// For a 16-row Galton board --> 16 possible exit slots
int num_bins = 0;
int histogram[17];
int histo_base_y = 480; // Just below the board
```

```
int histo_height = 60;  // Size of the histogram region
int bin_width = 38; // width of each bin

//constant values
fix15 fix15_1 = int2fix15(1);
fix15 fix15_minus2 = int2fix15(-2);
fix15 fix15_1over4 = float2fix15(0.25);

// Record the start time so can display "time since boot"
uint32_t start_time_us;

// -------------- DMA / Audio ---------------
#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7
#define SPI_PORT spi0
#define LED_pin 25

#define DAC_config_chan_A 0b0011000000000000
#define SINE_TABLE_SIZE 256
static int raw_sin[SINE_TABLE_SIZE];
static unsigned short DAC_data[SINE_TABLE_SIZE];
static unsigned short * address_pointer = &DAC_data[0];

static int data_chan;
static int ctrl_chan;
static dma_channel_config data_cfg;
static dma_channel_config ctrl_cfg;

// Forward declarations
void dma_trigger_beep(void);

// ---------------- Initialize Audio ----------------
void init_audio(void)
{
    // gpio_put(LED_pin, 1);
    spi_init(SPI_PORT, 20000000); // 20 MHz
    spi_set_format(SPI_PORT, 16, 0, 0, 0);
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
```

```
    gpio_set_function(PIN_CS,   GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK,  GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);

    for(int i=0; i<SINE_TABLE_SIZE; i++){
        raw_sin[i]   = (int)(2047.0 * sinf((float)i *
6.283f/(float)SINE_TABLE_SIZE) + 2047);
        DAC_data[i]  = DAC_config_chan_A | (raw_sin[i] & 0x0fff);
    }

    // Claim DMA channels
    data_chan = dma_claim_unused_channel(true);
    ctrl_chan = dma_claim_unused_channel(true);

    // Setup CONTROL channel
    ctrl_cfg = dma_channel_get_default_config(ctrl_chan);
    channel_config_set_transfer_data_size(&ctrl_cfg, DMA_SIZE_32);
    channel_config_set_read_increment(&ctrl_cfg, false);
    channel_config_set_write_increment(&ctrl_cfg, false);
    channel_config_set_chain_to(&ctrl_cfg, data_chan);

    dma_channel_configure(
        ctrl_chan,
        &ctrl_cfg,
        &dma_hw->ch[data_chan].read_addr,
        &address_pointer,
        1,
        false
    );

    // Setup DATA channel
    data_cfg = dma_channel_get_default_config(data_chan);
    channel_config_set_transfer_data_size(&data_cfg, DMA_SIZE_16);
    channel_config_set_read_increment(&data_cfg, true);
    channel_config_set_write_increment(&data_cfg, false);

    // Use DMA Timer0 for ~44 kHz
    dma_timer_set_fraction(0, 0x0017, 0xffff);
    channel_config_set_dreq(&data_cfg, 0x3b); // DREQ = TIMER0
```

```
    // Chain back to ctrl_chan
    channel_config_set_chain_to(&data_cfg, ctrl_chan);

    dma_channel_configure(
        data_chan,
        &data_cfg,
        &spi_get_hw(SPI_PORT)->dr,
        DAC_data,
        SINE_TABLE_SIZE,
        false
    );
    // gpio_put(LED_pin, 1);
}

/**
 * Plays one "beep" by starting the two DMA channels, waiting for
 * them to finish one cycle, then aborting.
 */
void dma_trigger_beep(void)
{
    dma_start_channel_mask(1u << ctrl_chan);
    //gpio_put(LED_pin, !gpio_get(LED_pin));

    dma_channel_wait_for_finish_blocking(data_chan);

    dma_channel_abort(data_chan);
    dma_channel_abort(ctrl_chan);
}

// ------------------- Helper to compare pegs -----------------
bool peg_compare(peg *peg1, peg *peg2){
    return peg2 == peg1;
}

// ------------------- Spawn a boid -----------------
void spawnBoid(boid* boid_obj){
    boid_obj->x = int2fix15(330);
    boid_obj->y = int2fix15(40);
}
```

```
// ------------------- Spawn the arena (all pegs) -----------------
void spawnArena() {
    num_pegs = 0;
    for (int row = 0; row < num_rows; row++){
        for (int col = 0; col <= row; col++){
            fix15 peg_x = int2fix15(320 + (col*38) - (row*19));
            fix15 peg_y = int2fix15(100 + (row*19));

            peg_obj_array[num_pegs].peg_x = peg_x;
            peg_obj_array[num_pegs].peg_y = peg_y;
            peg_obj_array[num_pegs].peg_row = row;
            peg_array[num_pegs]          = &peg_obj_array[num_pegs];
            num_pegs++;
        }
    }
}

// ------------------- Spawn boids -----------------
void spawnBoids() {
    for (int i = 0; i < max_boids; i++){
        boid_obj_array[i].x = int2fix15(320);
        boid_obj_array[i].y = int2fix15(50);
        boid_obj_array[i].vx = (rand() & 0xffff) - fix15_1;
        boid_obj_array[i].vy = 0;
        boid_obj_array[i].lastpeg = NULL;
        boid_array[i] = &boid_obj_array[i];
    }
}

// ------------------- Draw pegs -----------------
void drawArena() {
    for (int k = 0; k < num_pegs; k++){
        peg *thispeg = peg_array[k];
        fillRect(fix2int15(thispeg->peg_x-6), fix2int15(thispeg->peg_y-6),
12, 12, RED);
    }
}

// ------------------- Draw boids -----------------
void drawBoids(char c) {
```

```
    for (int k = 0; k < num_boids; k++){
        boid *b = boid_array[k];
        fillCircle(fix2int15(b->x), fix2int15(b->y), 4, c);
    }
}


void clearHistogram(){
    int bin0_pos = (320-38-38*((num_rows-1)/2)-19*((num_rows+1)%2));

    for(int i=0; i<num_bins; i++){
        if(histogram[i] > maxcount) {
            maxcount = histogram[i];
        }
    }

    float step = histo_height/maxcount;

    for(int i=0; i<num_bins; i++){
        int bar_h = histogram[i] * step;
        int bar_x = bin0_pos + i * bin_width;
        int bar_y = histo_base_y - bar_h;

        // fillRect(bar_x, bar_y, bin_width-1, bar_h, BLACK);
        drawRect(bar_x, bar_y, bin_width-1, bar_h, BLACK);
    }
}


// ------------------- Helper: reset histogram --> when shifting
potentiometer states -----------------
void reset_histogram() {
    for(int i = 0; i < num_bins; i++){
        histogram[i] = 0;
    }
    total_dropped = 0;
    maxcount      = 0;
}
```

```
// -------------------- Draw histogram -----------------
void drawHistogram() {
    // Suppose the bottom row of pegs is around y=400, and the screen is
480 tall
    // Adjust these values to fit your actual layout

    int bin0_pos = (320-38-38*((num_rows-1)/2)-19*((num_rows+1)%2));

    float step = histo_height/maxcount;

    for(int i=0; i<num_bins; i++){
        int bar_h = histo_height;
        int bar_x = bin0_pos + i * bin_width;
        int bar_y = histo_base_y - bar_h;

        fillRect(bar_x, bar_y, bin_width-1, bar_h, BLACK);
    }
    //comment out the for loop on top and comment in clear histogram in
walls and edges before loops

    // Find the largest bin count
    for(int i=0; i<num_bins; i++){
        if(histogram[i] > maxcount) {
            maxcount = histogram[i];
        }
    }

    // Avoid div-zero if no balls have dropped
    if (maxcount == 0) maxcount = 1;
    step = histo_height/maxcount;

    for(int i=0; i<num_bins; i++){
        int bar_h = (histogram[i] * step);
        int bar_x = bin0_pos + i * bin_width;
        int bar_y = histo_base_y - bar_h;

        drawRect(bar_x, bar_y, bin_width-1, bar_h, RED);
    }
}
```

```
// -------------------- Draw stats text -----------------
void drawStatsConst() {
  // Time since boot
  uint32_t now_us = time_us_32();
  float time_elapsed_s = (now_us - start_time_us) / 1e6f;

  char str_buffer[50];

  // 1) Position the cursor

  // 2) Set the text color
  setTextColor(WHITE);
  setTextSize(1);
  // 3) Format and print the string
  setCursor(5, 5);
  sprintf(str_buffer, "Total particles dropped: ");
  writeString(str_buffer);

  // Move down 10 or 12 pixels for the next line
  setCursor(5, 15);
  sprintf(str_buffer, "Active particles: ");
  writeString(str_buffer);

  // Another line
  setCursor(5, 25);
  sprintf(str_buffer, "Bounciness: ");
  writeString(str_buffer);

  setCursor(5, 35);
  sprintf(str_buffer, "Gravity: %.2f", fix2float15(gravity));
  writeString(str_buffer);

  setCursor(5, 45);
  sprintf(str_buffer, "Time elapsed:       s");
  writeString(str_buffer);

}
```

```
// -------------------- Draw stats text -----------------
void drawStatsDyn() {
    // Time since boot
    uint32_t now_us = time_us_32();
    float time_elapsed_s = (now_us - start_time_us) / 1e6f;

    char str_buffer[50];

    // Clear or overwrite the area here if needed (e.g. fillRect to erase
old text) -->  Not sure if we need to do this tbh

    // 1) Position the cursor

    // 2) Set the text color
    setTextColor2(WHITE, BLACK);
    setTextSize(1);
    // 3) Format and print the string
    setCursor(153, 5);
    sprintf(str_buffer, "%d  ", total_dropped);
    writeString(str_buffer);

    // Move down 10 or 12 pixels for the next line
    setCursor(107, 15);
    sprintf(str_buffer, "%d  ", num_boids);
    writeString(str_buffer);

    // Another line
    setCursor(80, 25);
    sprintf(str_buffer, "%.2f  ", fix2float15(bounciness));
    writeString(str_buffer);

    setCursor(83, 45);
    sprintf(str_buffer, "%.2f  ", time_elapsed_s);
    writeString(str_buffer);

  }
```

```
// -------------------- Update boids -----------------
void wallsAndEdges() {
    // clearHistogram();
    for (int i = 0; i< num_boids; i++){
        boid *boid = boid_array[i];
        fix15 x_speed = boid->vx;
        fillCircle(fix2int15(boid->x), fix2int15(boid->y), 4, BLACK);
        for(int j = 0; j < num_pegs; j++){

            peg *pg = peg_array[j];
            if (!(pg->peg_row > boid->lastpeg->peg_row+3)){

                // Compute x and y distances between ball and peg
                dx = boid->x - pg->peg_x;
                dy = boid->y - pg->peg_y;

                // Check approximate collision
                if ((absfix15(dx) <= (boid_r + peg_r)) && (absfix15(dy) <=
(boid_r + peg_r))) {
                    // Actual distance

                    distance = float2fix15(sqrtf(fix2float15(multfix15(dx,
dx) + multfix15(dy, dy))));

                    // alpha max beta min
                    // fix15 max_v = max(abs(dx), abs(dy));
                    // fix15 min_v = min(abs(dx), abs(dy));
                    // // fix15 max_v2 = multfix15(max_v2, max_v2);
                    // // fix15 min_v2 = multfix15(min_v2, min_v2);
                    // distance = float2fix15(multfix15(fix15_1, max_v) +
multfix15(fix15_1over4, min_v));

                    if (distance < (boid_r + peg_r)) {
                        // Normal vector
                        normal_x = divfix(dx, distance);
                        normal_y = divfix(dy, distance);

                        fix15 intermediate = multfix15(fix15_minus2,
(multfix15(normal_x, boid->vx) + multfix15(normal_y, boid->vy)));
```

```
                        // If ball velocity and normal are in opposite
directions
                        if (intermediate > 0){
                            // Teleport out of collision
                            boid->x = pg->peg_x + multfix15(normal_x,
distance + fix15_1);
                            boid->y = pg->peg_y + multfix15(normal_y,
distance + fix15_1);

                            // Collision update
                            boid->vx = boid->vx + multfix15(normal_x,
intermediate);
                            boid->vy = boid->vy + multfix15(normal_y,
intermediate);

                            // Strike a new peg?
                            if (!peg_compare(pg, boid->lastpeg)){
                                // beep
                                dma_trigger_beep();

                                // remove some energy
                                boid->vx = multfix15(bounciness,
boid->vx);
                                boid->vy = multfix15(bounciness,
boid->vy);
                                boid->lastpeg = pg;
                            }
                        }
                    }
                }
            }
        }

        // Check bottom
        if (hitBottom(boid->y)) {
            // Figure out which bin
            int xint = fix2int15(boid->x);

            int bin_index =
(xint-(320-38*((num_rows+1)/2)-19*((num_rows+1)%2)))/38;
```

```
            if (bin_index < 0) bin_index = 0;
            if (bin_index >= num_bins) {bin_index = num_bins - 1;};

            // Increment histogram

            histogram[bin_index]++;
            total_dropped++;

            // Respawn the ball
            boid->vy = 0;
            boid->y  = int2fix15(50);
            boid->x  = int2fix15(320);
            boid->lastpeg = NULL;
            boid->vx = (rand() & 0xffff) - fix15_1;
        }

        // Check top
        if (hitTop(boid->y)){
            boid->vy = -boid->vy;
        }
        // Check sides
        if (hitRight(boid->x) || hitLeft(boid->x)){
            boid->vx = -boid->vx;
        }

        // Gravity
        boid->vy = boid->vy + gravity;
        // Update position
        boid->x  = boid->x + boid->vx;
        boid->y  = boid->y + boid->vy;

        fillCircle(fix2int15(boid->x), fix2int15(boid->y), 4, GREEN);
    }
}

void board_reset(int old_num){

  for (int k = old_num; k < num_boids; k++){
    boid *b = boid_array[k];
```

```
    fillCircle(fix2int15(b->x), fix2int15(b->y), 4, BLACK);
    // b->x = 320;
    // b->y = 50;
  }
}



// ------------------ Button FSM ------------------
#define BUTTON_PIN 15
static int ui_state = 0;  // 0 = no adjustment, 1 = # boids, 2 =
bounciness
static bool last_button = true;
static uint32_t last_debounce_time_us = 0; //stores time when last valid
button press was acknowledged
#define DEBOUNCE_DELAY_US 20000 // 20 ms

// Check for a falling edge (press) with debounce
void check_button() {
    bool curr_val = gpio_get(BUTTON_PIN); // read pin (true if pulled up
--> voltage high --> not pressed ) false if pulled down
    if (!curr_val && last_button && //if pressed and the last button true
(not pressed)
        ((time_us_32() - last_debounce_time_us) > DEBOUNCE_DELAY_US))
    {
        last_debounce_time_us = time_us_32();
        // Cycle 0->1->2->0->... we shift states in this pattern
        ui_state++;
        if (ui_state > 2) ui_state = 0;
        reset_histogram();
    }
    last_button = curr_val; // set the condition of the current loop's
reading
}



// ================================================
// === users serial input thread
// ================================================
```

```
static PT_THREAD (protothread_serial(struct pt *pt))
{
    PT_BEGIN(pt);
    static int user_input ;
    PT_YIELD_usec(1000000) ;
    sprintf(pt_serial_out_buffer, "Protothreads RP2040 v1.0\n\r");
    serial_write ;

    while(1) {
        sprintf(pt_serial_out_buffer, "input a number in the range 1-15:
");
        serial_write ;
        serial_read ;
        sscanf(pt_serial_in_buffer,"%d", &user_input) ;

        if ((user_input > 0) && (user_input < 16)) {
          color = (char)user_input ;
        }
    }
    PT_END(pt);
}

// ==================================================
// === Animation on core 0
// ==================================================
static PT_THREAD (protothread_anim(struct pt *pt))
{
  PT_BEGIN(pt);
  static int begin_time ;
  static int spare_time ;

  // Initialize boids & arena
  num_bins = num_rows+1;
  spawnBoids();
  spawnArena();
  drawStatsConst();

    // ADDED: At reset => max # boids
    num_boids = max_boids;
    reset_histogram();
```

```
 while(1) {
     begin_time = time_us_32() ;

     // 1) Check button FSM
     check_button();
     potentiometer = (adc_read() >> 4);

   // 3) Adjust parameter based on ui_state
   if (ui_state == 1) {
       drawBoids(BLACK);
       int old_num = num_boids;
       // Adjust number of boids
       int new_boids = (potentiometer * max_boids) / 256;
       if (new_boids != num_boids) {
           // Don't let it be 0 if you prefer at least 1
           num_boids = (new_boids < 1) ? 1 : new_boids;
       }
   }
   else if (ui_state == 2) {
       // Adjust bounciness [0.1..1.0], check what the min bounce can
be??
       fix15 new_bounciness = float2fix15(
           0.1f + 0.9f * (potentiometer / 255.0f)
       );
       if (new_bounciness != bounciness) {
           bounciness = new_bounciness;
       }
   }
   // if ui_state == 0 => do nothing with pot
   //   clearHistogram();
     // Update boids

     wallsAndEdges() ;

     // Redraw pegs, histogram, stats
     drawArena();
     drawStatsDyn();
     drawHistogram();
```

```
      // Maintain frame rate
      spare_time = FRAME_RATE - (time_us_32() - begin_time) ;
      if (spare_time < 0) {
          gpio_put(LED_pin, !gpio_get(LED_pin));
      }
      PT_YIELD_usec(spare_time) ;

  }
  PT_END(pt);
}

// ====================================================
// === Animation on core 1
// ====================================================
static PT_THREAD (protothread_anim1(struct pt *pt))
{
    PT_BEGIN(pt);
    static int begin_time ;
    static int spare_time ;

    while(1) {
        begin_time = time_us_32() ;

        // Update boids
        // wallsAndEdges() ;

        spare_time = FRAME_RATE - (time_us_32() - begin_time) ;
        if (spare_time > 0) {
            PT_YIELD_usec(spare_time) ;
        }
    }
    PT_END(pt);
}

// ========================================
// === core 1 main -- started in main below
// ========================================
void core1_main(){
    pt_add_thread(protothread_anim1);
```

```
    pt_schedule_start;
}

// ========================================
// === main
// ========================================
int main(){
    stdio_init_all();
    initVGA();
    init_audio();

    adc_init();
    // Make sure GPIO is high-impedance, no pullups etc
    adc_gpio_init(26);
    // Select ADC input 0 (GPIO26)
    adc_select_input(0);


    // Record start time
    start_time_us = time_us_32();

    // start core 1
    multicore_reset_core1();
    multicore_launch_core1(&core1_main);

    gpio_init(LED_pin);
    gpio_set_dir(LED_pin, GPIO_OUT);


    // ADDED: Configure button input
    gpio_init(BUTTON_PIN);
    gpio_set_dir(BUTTON_PIN, GPIO_IN);
    gpio_pull_up(BUTTON_PIN);


    // Add threads
    pt_add_thread(protothread_serial);
    pt_add_thread(protothread_anim);
    pt_add_thread(protothread_anim1);
```

```
    // Start scheduler
    pt_schedule_start;
}
```