

ECE 5730 Lab 2 Report

Galton Board



Amber Li | Ivan Huang | Zarif Karim
10.22.2026

Maximum # balls: 16,900
Clock Frequency: Baseline: 150 MHz → Overclocked: 350 MHz

Introduction

Imagine gambling your ECE 4760 grade for a Dunkin' pumpkin spice latte. This is what dropping balls down a galton board feels like!

That's exactly what we built in this lab: a digital Galton board in which a user-defined amount of balls drop down from the top of a board of 136 pegs among 16 rows that make a dropping thud sound drawn out on a 640x480 VGA monitor. Ball motion is based on real collision physics, travelling down the board dropping among slots between pegs such that a histogram of a bell curve is generated.

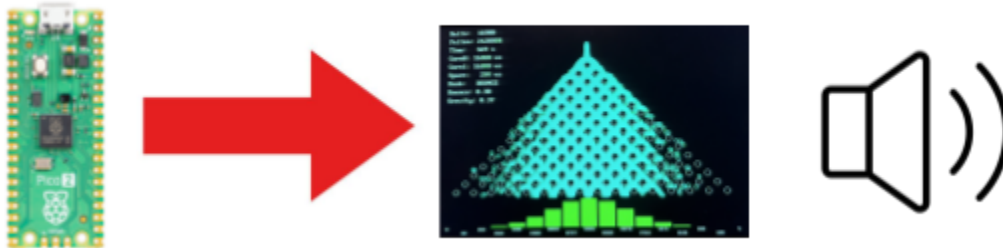


Figure 1: Lab Overview

Every time a ball hits any peg, the DMA channel plays a short “thunk” sound through the SPI controlled DAC without any CPU intervention, freeing more resources to add more optimizations and allow more balls to undergo physics calculations on screen. A rotary encoder, with push button functionality, allows the user to change the number of balls, “bounciness” of the balls, and the gravity at which the ball falls at (ECE 5730 specific).

Starting point: first measured version handled 5,000 balls at the default 150 MHz clock. After completion of base functionality, we then implemented the following:

- O(1) runtime peg lookup
- Splitting work across both cores
- Precomputed ball shape
- Hollow
- Cheaper collision math
- Parallel screen clearing
- Overclocked to 350 MHz
- Squeezing each ball into 12 bytes
- Moving hot code from flash into SRAM

Final result: 16,900 balls at 60 fps, with about 600 μ s of spare time per frame

Mathematical Background

Galton Board Statistics

A ball hits only 1 peg in each row. From that one bounce, it has $\frac{1}{2}$ chance of traveling left, and $\frac{1}{2}$ chance of traveling right. Upon bouncing off the final row of an “n” row galton board, the ball lands in a bin. This bin that the ball lands in is determined by how many times the ball bounced to the left and right 😊. To simplify this...

$$P(K = k) = C(n, k) / 2^n, \quad k = 0, 1, \dots, n$$

Equation 1: Bernoulli Trial after n rows

$C(n, k)$ represents any row “n” on the Galton Board.

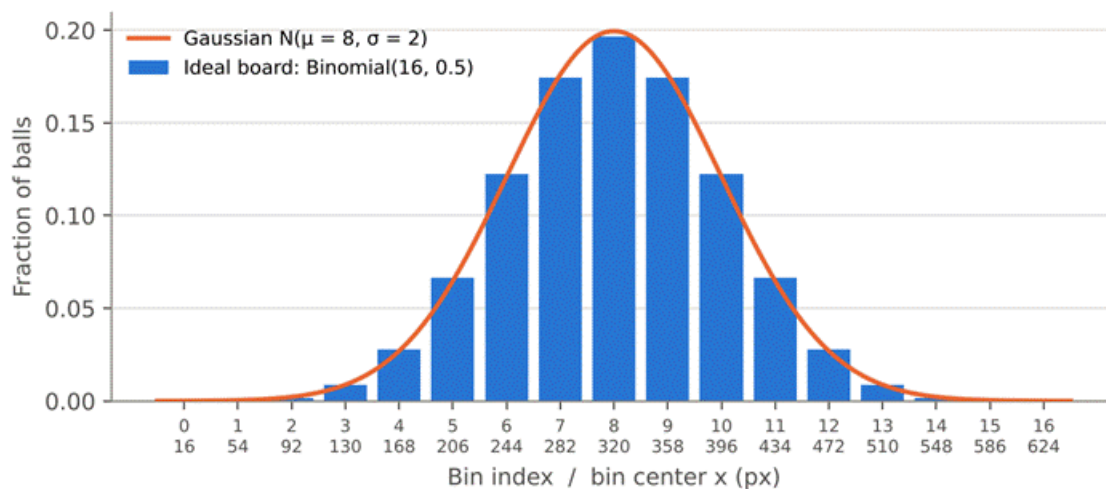


Figure 2: Ideal 16-row Galton board: Binomial(16, 0.5) bin probabilities (bars) and the approximating Gaussian with $\mu = 8$, $\sigma = 2$ bins (line). The second row of x-axis labels gives each bin's center on the screen.

The ideal Galton Board’s bin distribution produces a Gaussian distribution. Each ball has a 50% chance of going left or right at each peg, and since there are more paths to get to the middle bins, more balls end up there. The central limit theorem tells us that the sum of many independent, identically distributed random variables should approach a normal distribution with larger numbers of variables. Since each ball in an ideal Galton Board is iid with two equally possible outcomes per trial and a fixed number of trials, the distribution is binomial. As the number of rows grows, there are more ways to get to the central bins compared to the edges, creating the Gaussian distribution.

Bouncing Ball Physics

At the most foundational levels the following is used to compute kinematics in our system:

- Pixels = distance units
- Frames = time

In each frame, balls move by a velocity in the x direction, then accelerate by gravity in the y direction. The Galton board can be modeled as many balls bouncing off of round pegs. The pegs are modeled as balls with infinite mass, and stay stationary. From the collision physics webpage, the general equation for collision then simplifies to the following:

$$\Delta \mathbf{V} = \left[\frac{2m}{m+M} \left(\frac{\mathbf{r}_m - \mathbf{r}_M}{\|\mathbf{r}_m - \mathbf{r}_M\|} \cdot (\mathbf{v}_i - \mathbf{V}_i) \right) \right] \frac{\mathbf{r}_m - \mathbf{r}_M}{\|\mathbf{r}_m - \mathbf{r}_M\|}$$

Equation 2: Velocity of a Ball Bouncing Off a Round Peg

which can be rewritten as

$$\vec{v}' = \vec{v} - 2(\vec{v} \cdot \hat{n}) \hat{n}$$

Equation 3: General equation of previous Ball Velocity equation

where $\hat{n}$ is the unit normal vector from the peg's center to the ball's center, found by calculating the x and y distances dx and dy:

$$\hat{n} = \frac{(dx, dy)}{\sqrt{dx^2 + dy^2}}$$

Equation 4: Unit normal vector equation

Since the pegs are stationary, only the ball's velocity changes when a ball bounces off a peg. The collision reflects the velocity component directed along the normal vector, while keeping the tangential component. A coefficient of restitution is multiplied to the resulting velocities to model energy loss of a non-perfectly elastic collision. Energy loss and the thunk sound only happen when the ball hits a new peg, otherwise a ball rolling along one peg would keep losing speed and retriggering the sound.

Fixed Point Arithmetic

Although the Pico 2's CPU contains a Floating Point Unit (math block that works on decimals), it is actually still faster to use fixed point arithmetic. Running through the ALU required less steps to compute on purely binary numbers.

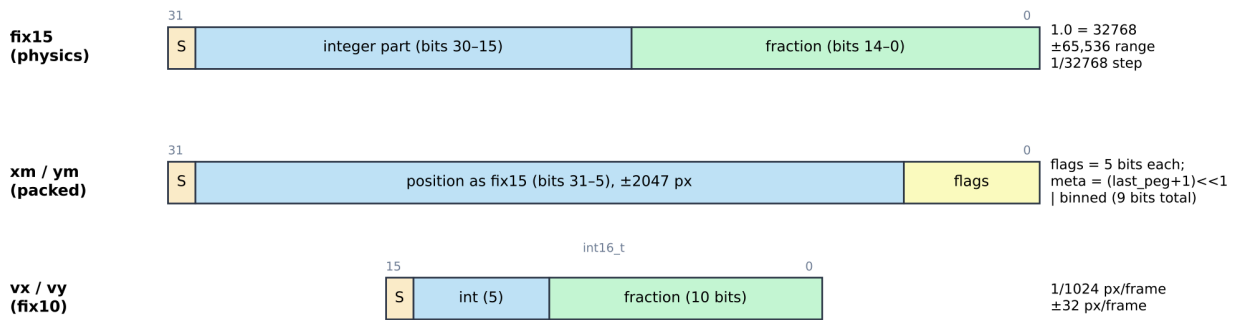


Figure 3: Fixed-point formats. Top: fix15 (signed 17.15) used for every physics calculation. Middle and bottom: the packed position and velocity formats inside our 12-byte ball struct.

Physics calculations were done using what we call “fix15.” In this case, it still looks as if it is a signed 32 integer, but it's actually split into 3 sections: MSB still signifies positive/negative, but now the 15 in fix means at the 15th bit, bits 14 $\rightarrow$ 0 represents bits of a fractional number. The remaining 16 bits 30 $\rightarrow$ 15 represent a 16 bit binary number.

Design and Testing Methods

Rotary Encoder

A rotary encoder was best suited for our program. It is better than keypad buttons as it provides discretized incremental control of parameters. Compared to our birdsong pitch slider which incorporated a continuous range of values to change, we have less fine grain adjustments to perform when changing parameters on our galton board, so individual turns of the knob allowed for precisely controlled adjustment actions. Due to the nature of the knob hardware, though, some detection code was required. Below is the internal of a rotary encoder:

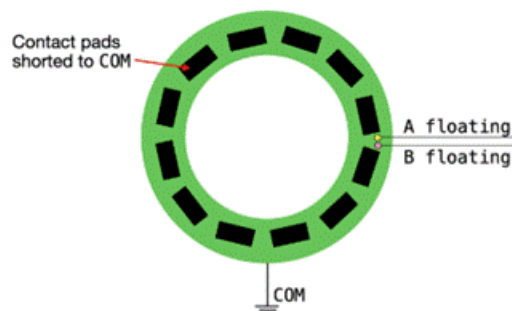


Figure 4: Rotary encoder internal mechanism. (Source: V. Hunter Adams, "How does a rotary encoder work?")

Inside is a rotating disk with evenly spaced contact blocks in a rotating circle connected to some common ground. When the internal blocks touch either A or B, they pull up to 1. Notice how A and B are either above or below each other depending on which direction you rotate. We have two different sequences of which A and B can be contacted in:

- A then B (AB = 10)
- B then A (AB = 01)
- Nothing (AB = 00)

To tell the difference between a counterclockwise and a clockwise rotation, we needed to detect these sequences of signals and translate them into either a CW or CCW rotation. In essence, we must *decode* the rotary *encoder*. Get it? Lol.

Below is an oscilloscope reading of rotary encoder use:

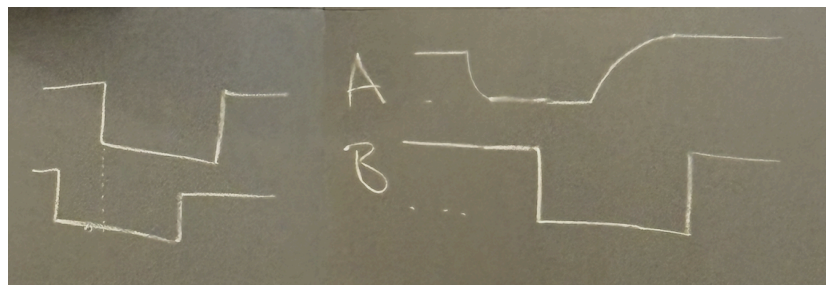


Figure 5: Rotary Encoder Signal Waveform Shape

The channels A and B are connected to GPIOs which are configured as inputs with pull-up resistors. The rotary encoder also has a push button function, which we tie to another GPIO pin configured as an input but with a pull-down resistor, since it is active high.

Rotary Encoder Timer Interrupt

We decided to decode inside of an interrupt. This allows us to only perform decoding when a signal edge is detected from the rotary encoder, rather than polling the CPU frequently and cutting into processing time.

Our first version of decode just used a call to the ISR on a rising or falling edge of ROT_A, then checks if ROT_A = ROT_B which signals that the rotary knob is mid turn. If ROT_A = ROT_B, that means B must have made contact first before A got triggered, indicating counterclockwise rotation. Otherwise, A is pulled high while B is still low, which is the first position of the clockwise rotation. However, we noticed that this resulted in multiple increments occurring as the knob was turned, likely due to mechanical bouncing so we had to redesign a debouncing decoder.

Our final version triggers on both the rising and falling edges of A and B.

```
gpio_set_irq_enabled_with_callback(ROT_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true, &rot_ISR);
gpio_set_irq_enabled(ROT_B, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true);
```

Right before the ISR callback is entered, the GPIOs are read to store a previous state of rotary A and B in a 2-bit form AB. Once an edge is detected and the callback is entered, they are read again to get a current state value. The previous and current states are concatenated into a 4-bit number of the form [prev_A, prev_B, curr_A, curr_B], which is used to search a lookup table to provide the result of whether we should accumulate 0, +1, or -1. The states in the lookup table are configured so that the four states that make up a clockwise rotation all provide +1, while the four states that make up a counterclockwise rotation give -1. Thus, when the current state is detected to be back in the reset position, we check whether rot_accum has reached 4 or -4, and signify a CW or CCW signal, respectively.

```
static volatile uint8_t rot_prev_state = 3 ; // (A<<1)|B ; 3 = rest (both high)
static volatile int8_t  rot_accum = 0 ;      // steps taken since last rest

// Gray-code step table, index = (prev<<2)|curr: +1 CW, -1 CCW, 0 = no step or bounce
static const int8_t rot_table[16] = {
/* prev=0(00) */ 0, -1, +1, 0,
/* prev=1(01) */ +1, 0, 0, -1,
/* prev=2(10) */ -1, 0, 0, +1,
/* prev=3(11) */ 0, +1, -1, 0
};

...

// Fires on every edge of A or B. A click counts only after 4 steps back at
// rest, so contact bounce (+1 then -1) cancels out.
void rot_ISR(uint gpio, uint32_t events)
{
    uint8_t curr_state = (gpio_get(ROT_A) << 1) | gpio_get(ROT_B) ;
    rot_accum += rot_table[(rot_prev_state << 2) | curr_state] ;
    rot_prev_state = curr_state ;

    if (curr_state == 3) { // landed back at rest (a full click, or none)
        if (rot_accum >= 4) rot_click(+1) ; // CW
        else if (rot_accum <= -4) rot_click(-1) ; // CCW
        rot_accum = 0 ; // discard partial turns / bounce
    }
}
```

Because the variables in an ISR can be modified at any time, they are made volatile which tells the compiler to manually retrieve the value from memory each time. We had an extra GPIO set to high once entering the interrupt, and probed the pin to see that it was firing each time we turned the knob and made contact with pin A or B.

Additionally, we added some hardware debouncing to our encoder channel signals in the form of capacitors to clear high frequency noise found whilst turning the rotary encoder. The added delay makes it so voltage does not change instantaneously, so rapid fluctuations due to mechanical bouncing do not as easily get translated to multiple high and low signals when reading from the GPIOs.

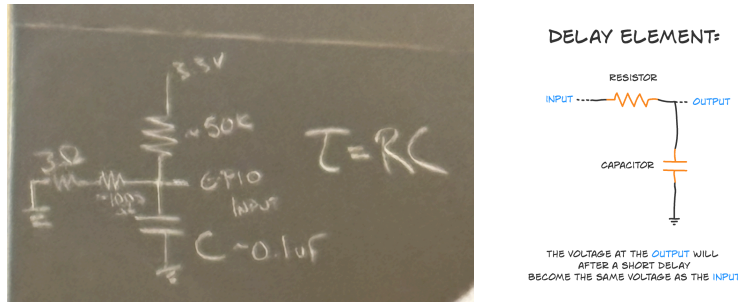


Figure 6: RC Delay

(Source: Lecture & <https://www.build-electronic-circuits.com/rc-delay-element/>)

This lets us modify the parameters of our Galton board, such as changing the number of balls being simulated by +/- 100, the bounciness parameter by +/- 0.5, or the gravity parameter by +/- 0.05. We use the rotary encoder's push button functionality to switch between editing parameters. For the push button, we implemented polling so that the button is sampled every frame, or every $1/60 \sim 16.67\text{ms}$ at 60 fps, which is enough time to avoid mechanical contact bouncing. We also have a `sw` and `sw_prev` state tracking so holding the button does not repeatedly switch parameters. Finally, when a Galton board parameter is changed, we reset the count of the total balls that have fallen through and the histogram.

DMA Audio

We've outsourced audio output computation to travel directly from memory out to the DAC. This is done by use of a Direct Memory Access (DMA) controller, which has data and control channels. It's like playing "ping-pong" at your internship with other interns! (-- Hunter Adams 2026). No CPU intervention is required, making space for even more computation.

On startup, we have an `initPegSound()` function that generates an 800 Hz sine wave with the appropriate short linear beginning and ends of the audio envelope to make a "thunk" sound, and appends the DAC configuration bits to each sample (Channel A, 1x gain, enable output, 12-bit audio sample). When a ball hits a new peg, the `playPegSound()` function checks if a sound is already playing by checking whether the data and control channels are busy. If they are, the function immediately returns, but if not, it starts the peg sound.

The control channel resets the data channel read pointer, so a new sound begins at the beginning of the waveform. The data channel walks through the audio array, kinda like our DAC data array from lab 1, writing to the SPI TX register, which then transmits the digital samples to the DAC and then to the audio jack.

Rather than have a CPU interrupt set the frequency that samples are transmitted across SPI to the DAC, the DMA audio hardware sets its own pacing using a timer configured to transfer samples at 50 kHz. The timer is set using a fraction of the system clock over the target DAC sample rate.

```
int snd_timer = dma_claim_unused_timer(true) ;  
dma_timer_set_fraction(snd_timer, 1, (uint16_t)(clock_get_hz(clk_sys) / SOUND_FS)) ;
```

In terms of chaining and triggering, we just start the control channel and it runs. It will repeat the process of triggering the control and resetting the pointer so playback of the audio is restartable without manually rewriting the audio sample. These design choices save a lot of computation time as the CPU does not have to handle 50,000 audio interrupts and generates a sine wave envelope once at the beginning.

VGA Driver (Pre optimization)

In order to display graphics, we developed both software and hardware for this purpose. We not only built upon a VGA driver provided by Hunter Adams, but also we soldered a custom PCB that streamlined required VGA connections. Below is a snapshot of the PCB used:

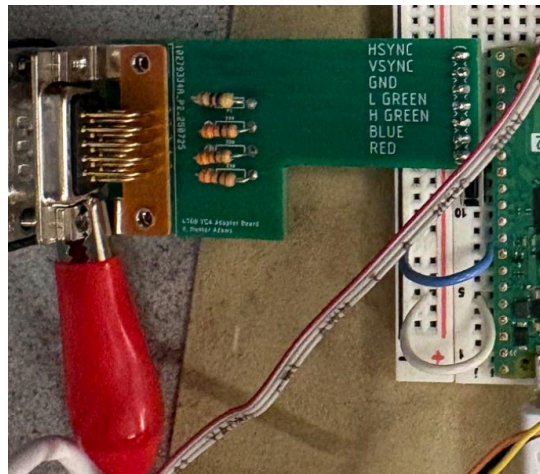


Figure 7: VGA PCB

Afterwards we then built upon a given VGA demo. We were given a double-buffered version of the course VGA driver. Three PIO state machines generate hsync, vsync, and the 4-bit color output, and a DMA chain feeds them pixel data. At every vsync time, the DMA chain tells the buffer we just finished drawing, points at the other buffer, and sets a flag. This gives both cores about $16,667 \mu\text{s}$ ($\sim 16 \text{ ms}$) to draw each frame. If they run late, the half-drawn buffer gets shown anyway so the frame is torn rather than slowed down.

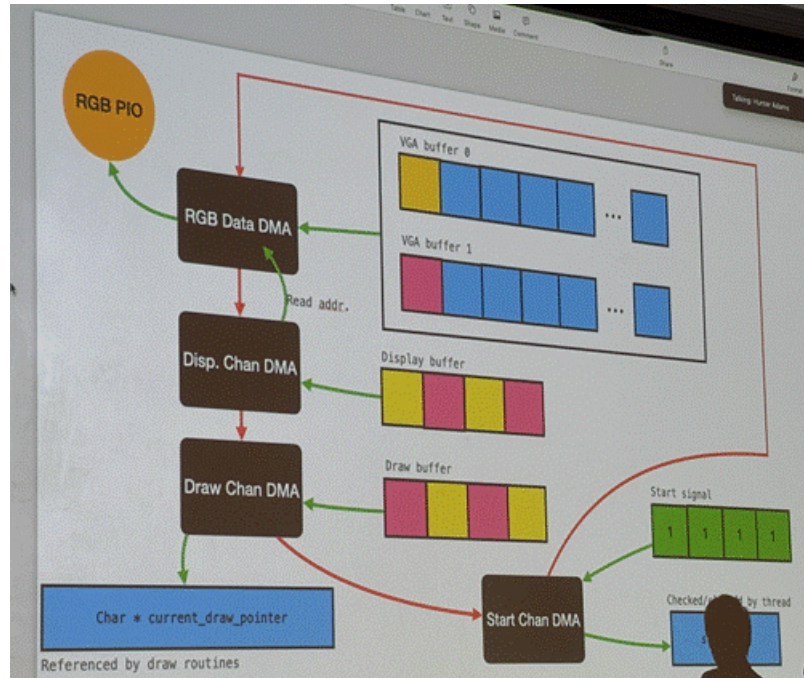


Figure 8: VGA Controller Block Diagram

In the initial VGA demo folder was a working double-buffered, 16 bit color driver where two balls are bouncing inside a rectangle area.

Frame buffer layout works like this. One frame is 640 by 480 by 4 bits. Stores row by row. That ends up being 320 bytes per row. Two pixels share a byte, one pixel takes the lower half and other takes the higher half. This is important late in our optimizations, such as our ball sprite layout, figuring out how to write whole bytes to draw two pixels simultaneously. Another reason this is important is because we have two buggers, about 300 of the 520 kB of SRAM is used, causing RAM to become another limiting factor of maximum balls.

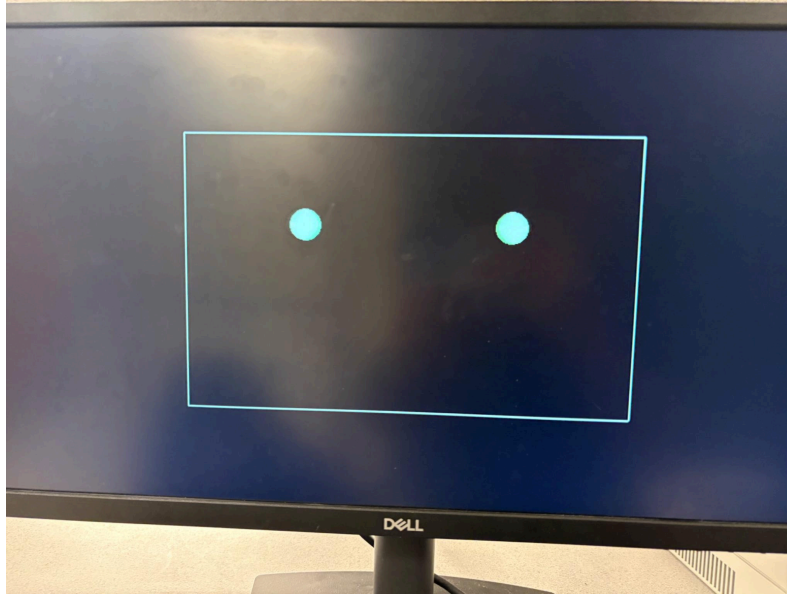


Figure 9: VGA Driver Demo

The first thing we drew was text. To test the rotary encoder, we drew out a live counter with `drawTextVGA437()`. Then, we got one ball bouncing off of one peg. From there, we added more pegs to eventually fill up the screen. We then went from one ball to an array of balls.

In total, the drawing functions we used were clearing, drawing filling, drawing pixels, and drawing text all with a black background. Digging into these functions we found heavy computing, like multiple square roots, so much that we created custom functions later to draw on the screen. The given library was still enough to complete our baseline design, though!

Ball and Peg Physics Implementation

Each ball's motion involves x , y position, vx , vy velocity, and gravity. At every ball update, the ball's position gets updated by $x = x + vx$ and $y = y + vy$, and gravity gets applied by $vy = vy + gravity$. Fast balls move several pixels per frame, enough to even skip past pegs, so we split each frame's motion into substeps, about 4 px.

The board contains 16 rows of pegs for a total of 136 pegs. Pegs are implemented as a circle slightly larger than balls with a fixed x and y coordinate position and no velocity. The balls' interactions with pegs are calculated through collision algorithms. When a ball nears a peg, we calculate the distance between their centers and see if that distance is less than the sum of their radii.

```
fix15 dx = x - peg_x[peg] ;  
fix15 dy = y - peg_y[peg] ;
```

An initial optimization included a cheap bounding-box where if the dx and dy distances were too far away then we guarantee there will not be a collision and we skip further calculations.

```
if ((absfix15(dx) < COLLIDE_DIST) && (absfix15(dy) < COLLIDE_DIST)) {
```

Once the ball is determined to collide with a peg, the collision calculations determine the direction of the ball's bounce. The normal vector is computed to tell which direction the ball should bounce relative to the peg's surface.

```
float inv_dist = 1.0f / sqrtf(fix2float15(dist2)) ;
fix15 normal_x = (fix15)(dx * inv_dist) ;
fix15 normal_y = (fix15)(dy * inv_dist) ;
```

To handle both cases of balls approaching the peg vs leaving the peg, we take a dot product between the ball's velocity v and the normal vector n , so if $v \cdot n < 0$, the ball's velocity gets reflected since the ball is headed towards the peg. Otherwise, there is no reflection so a ball traveling away does not bounce back into the peg. The reflected velocity also gets scaled by the bounciness parameter on each new peg bounce.

```
vx = multfix15(bounciness, vx) ;
vy = multfix15(bounciness, vy) ;
```

Once a ball passes the bottom row of the Galton board, we add its data to the histogram and respawn it at the top of the board with a new randomized initial position and velocity.

```
if (y > int2fix15(SCREEN_H)) {
    spawnBall(b) ;
    return ;
}
```

As soon as the ball's sprite completely drops below the final row of pegs, it is counted. The bin of which a ball is counted towards depends on which gap (between two pegs) of the final row the ball drops between (balls typically just drift to the side, so we ensure counting before they seemingly drop into another bin) to determine which gap a ball fell through, we perform a simple computation:

```
int bin = (xi < 0) ? 0 : (xi / PEG_HORZ_SEP) + 1 ; // left of it = bin 0
```

That is, we divide the balls x coordinate by the separation, finding its index based on the distance it is from the left most bin.



Figure 10: Galton Board Histogram

The table below shows the parameters we used to draw and initialize the Galton Board.

Parameter	Value	Notes
Ball radius / peg radius	4 px / 6 px	contact distance = 10 px
Row spacing / peg spacing	19 px / 38 px	each row shifts a ball ± 19 px

Rows / pegs / bins	16 / 136 / 17	row r has $r + 1$ pegs
Top peg / bottom row	$(320, 100) / y = 385$	histogram below $y = 400$
Spawn point	$x = 320 \pm 0.5 \text{ px}, y = 50$	50 px above the top peg
Gravity	0.37 px/frame^2 (0.05–1.00)	adjustable, our 5730 state
Bounciness	0.30 (0.00–1.00)	lowered from 0.5 in Week 2

Table 1: Galton Board Parameter Values

Final Wiring

Below is the table of connections:

Signal	RP2350 pin	Notes
VGA Hsync / Vsync	GPIO 16 / 17	PIO state machines
VGA Green lo / Green hi / Blue / Red	GPIO 18 / 19 / 20 / 21	$470 \Omega / 330 \Omega / 330 \Omega / 330 \Omega$
Encoder A / B	GPIO 12 / 11	internal pull-ups, $0.1 \mu\text{F}$ to GND each, edge IRQ
Encoder push button	GPIO 10	active high, internal pull-down, polled
MCP4822: CS / SCK / MOSI	GPIO 13 / 14 / 15	spi1, fed by DMA
On-board LED	GPIO 25	turns on when a frame misses 60 fps

Table 2: RP2350 Wiring Connections

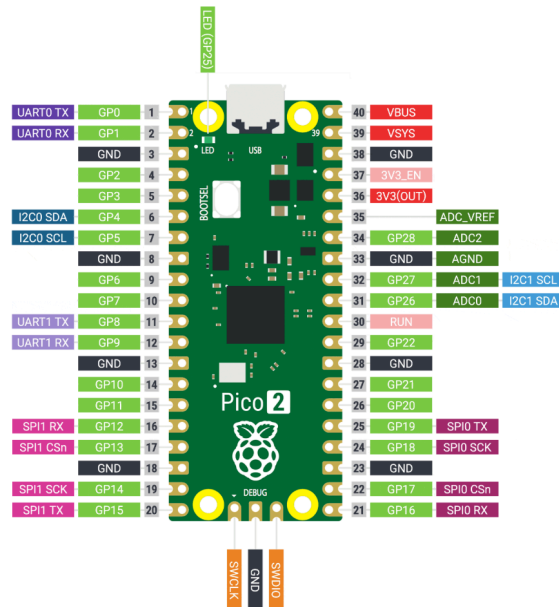


Figure 11: RP2350 Pinout

Optimizations

OPT	Change	Bottleneck it fixed	Measured result
1	O(1) nearest-peg lookup	136 peg tests per ball per substep	Made multi-ball possible (not measured alone)
2	Split balls across both cores	Core 1 sat idle	$\sim 2\times$ compute (not measured alone)
3	Precomputed ball sprite + float literals	fillCircle for every ball	5,000 balls @ 150 MHz
4	Cheaper collision math	sqrt and 64-bit software divides	5,200 balls @ 150 MHz
5	Parallel clear + auto split	Core 0 cleared alone; fixed split	Shipped with OPT 6
6	Overclock 150 $\rightarrow$ 300 $\rightarrow$ 350 MHz	CPU time	300 MHz: 3,150 μ s spare at 10k; 350 MHz: 13,000 at 0 μ s spare
7	Ball struct 24 $\rightarrow$ 20 $\rightarrow$ 16 $\rightarrow$ 12 bytes	RAM (linker overflow above ~ 8 k balls)	10,300 (20 B), 12,500 (16 B), RAM ceiling ~ 17 k (12 B)
8	Hot code in SRAM	Shared 16 KB flash cache; memset calls	13,000 @ 0 μ s $\rightarrow$ 16,000 @ ~ 600 μ s spare

Table 3: List of Optimizations

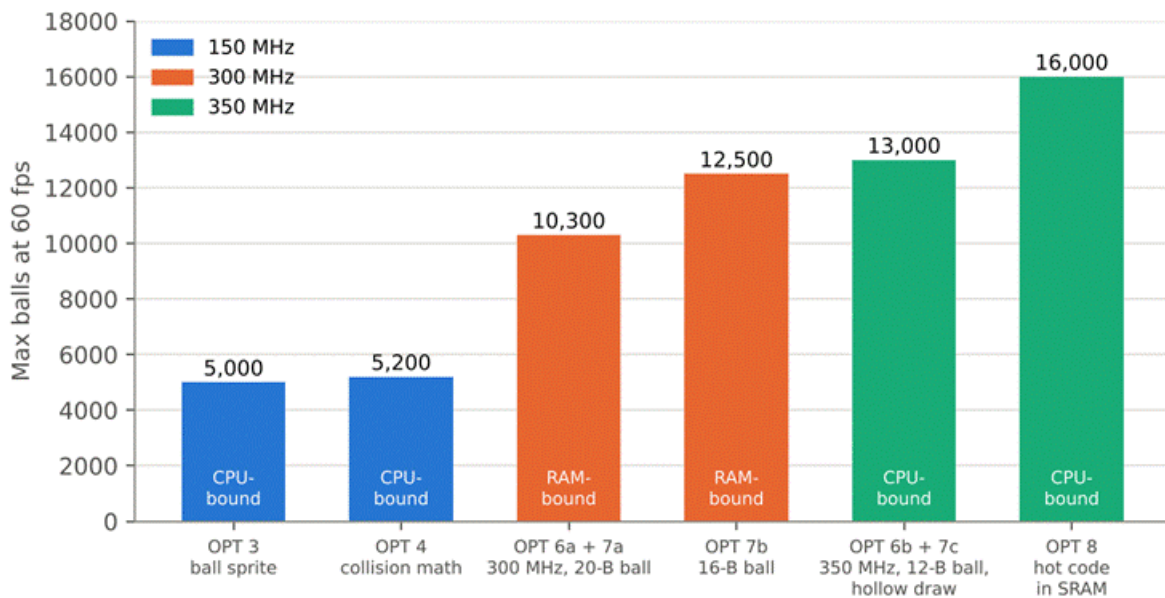


Figure 12. Maximum balls at 60 fps after each measured optimization, colored by system clock. The label inside each bar is what limited us at that step.

O(1) Peg Lookup

Originally, this lab's collision pseudocode checked against ALL pegs. Checking one ball against a total of 136 pegs at every frame is both wasteful and time consuming. Lecture suggested limiting a ball to just its nearby pegs, for example the row it's in. But we went even further.

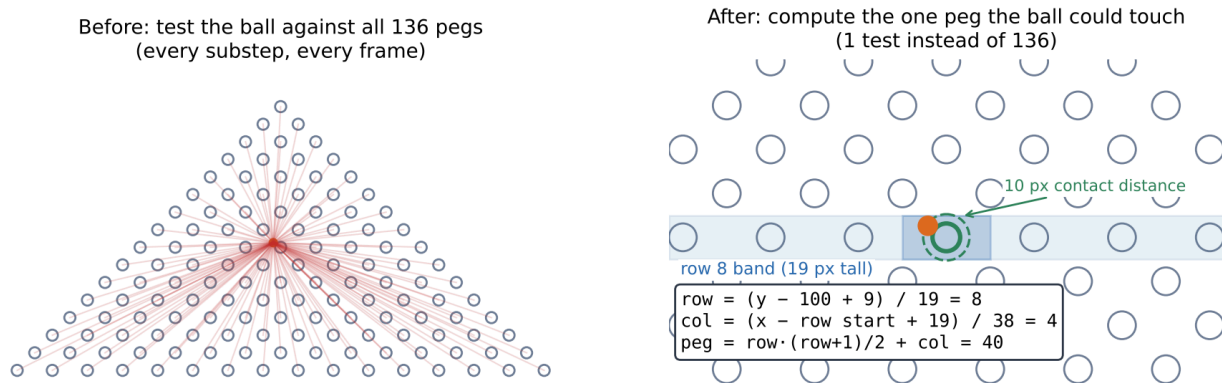


Figure 13: Nearest Peg Lookup Optimization

We determined that a ball touched a peg if their centers are less than 10 pixels apart. At any given moment. The only peg a ball would possibly touch is that of the one nearest to it. We created a new function to compute the peg from a ball's coordinates.

We effectively convert 136 checks into one check, which is 136 times less collision calculations.

Overclocking

At 150 MHz we ran out of CPU time at 5,200 balls, to push the max ball count on screen further, we overclocked the Pico. The RP2350 doesn't raise its core voltage automatically when overclocked and the default 1.10 V isn't enough for our desired clock frequency. To support overclocking of the processor, we first need to raise the voltage level to the max that the RPi allows: 1.30V.

```
vreg_set_voltage(VREG_VOLTAGE_1_30) ;
```

Changing system clock configurations to raise the frequency from 150MHz to 350MHz.

```
124 void clock_configure_undivided(clock_handle_t clock, uint32_t src, uint32_t auxsrc, uint32_t
    src_freq) {
125     clock_configure_internal(clock, src, auxsrc, src_freq, 1u <<
        CLOCKS_CLK_GPOUT0_DIV_INT_LSB);
126 }
```

Example application from RP2350 Datasheet:

```
100 // CLK SYS = PLL SYS (usually) 125MHz / 1 = 125MHz
```

```
101 clock_configure_undivided(clk_sys,
```

```
102 CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
```

```
103 CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS,
```

```
104 SYS_CLK_HZ);
```

The PLL was also reconfigured to a VCO (voltage-controlled oscillator) 1050 MHz / 3 / 1 to allow the system clock to be 350MHz.

```
pll_init(pll_sys, 2, 1050 * MHZ, 3, 1) ;
```

Changing the system clock also means changes to VGA driver are needed. The edits made to the .pio files in the VGA driver are described below:

- hsync.pio & vsync.pio - changed the clock divider to 14; initially $150\text{MHz}/6 = 25\text{MHz}$; now overclocked to $350\text{MHz}/14 = 25\text{MHz}$
 - `sm_config_set_clkdiv(&c, 14)`
- rgb.pio - changed pixel1hold from 5 -> 13 and pixel2hold from 3 -> 11. The VGA driver uses D as the chosen pixel clkdiv, so for a 25MHz pixel clock and 150MHz sys clk, $D = 6$. The general rule for the pixel hold cycles is $\text{pixel1hold} = D - 1$, $\text{pixel2hold} = D - 3$. This is how we determined what D should be for the system overclocked to 350MHz.
- This state machine runs at exactly vid_freq MHz, and pixel1hold & pixel2hold are used to manually stall the exact duration to achieve a 25MHz pixel clock
- Output 4 pins of color R G G B

Getting exactly 350 MHz while keeping VGA at 25 MHz and audio at 50 kHz

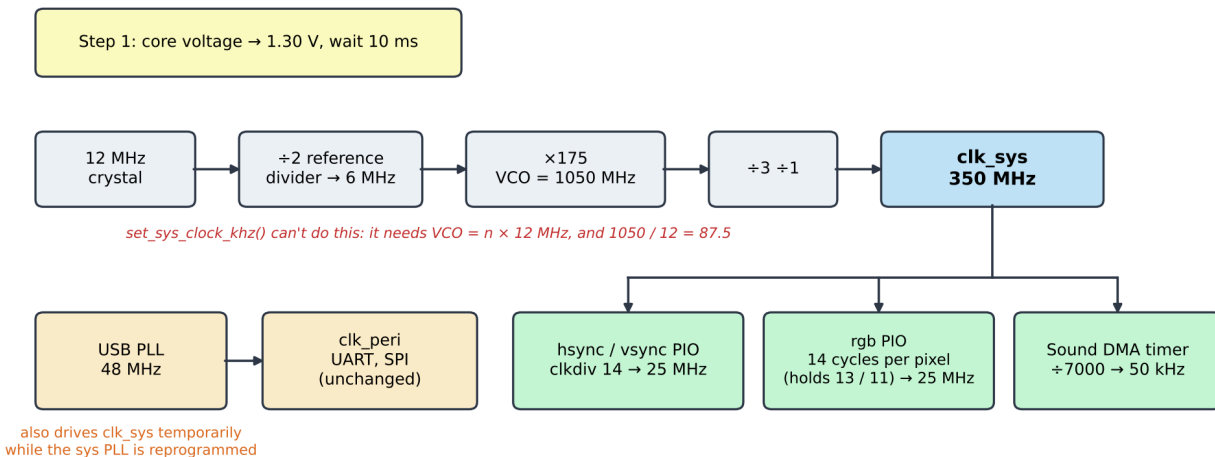


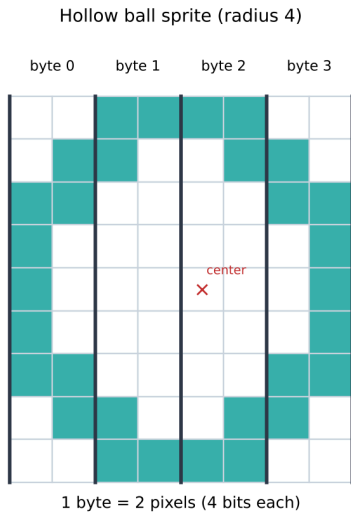
Figure 14: VGA Driver Clock and Timing Diagram

Precomputed Ball sprite

After collision became cheap, drawing graphics now seemed heavier in computation than before. The problem we had was that “fillCircle()” recomputes all of a shape on every single call. Within this function are a couple of square roots, another function call “drawHLine()”

initBallSprite() computes each row’s half-widths once at startup, and drawBall() writes the rows straight into the current frame buffer. Each byte then holds 2 4-bit pixels, then we fill whole bytes and handle a single nibble only at a row’s ends. In other words...

Our Galton ball became a hollow ring, greatly reducing graphic computation cost!



Before: fillCircle() for every ball, every frame
5 software square roots + 10 drawHLine() calls,
each with range checks

After: initBallSprite() computes each row's half-widths once
drawBall() writes each row as whole bytes straight into
the frame buffer (two short runs per row for the ring);
balls at the screen edge fall back to drawCircle()

row	outer half-width	hole half-width
0	4	3
± 1	4	3
± 2	4	2
± 3	3	1
± 4	2	0 (solid)

Figure 15: Ball Sprite Pixels

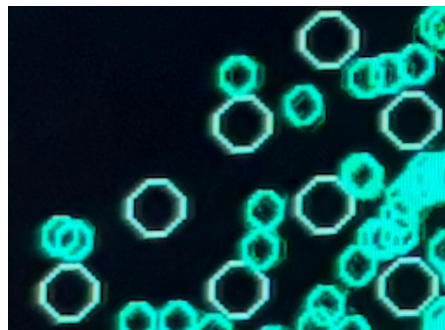


Figure 16: Optimized Balls and Pegs on Screen

At the same time we fixed the double literal in float2fix15. Both balls and pegs looked almost like octagons! However, they still retain the properties of a circle ball with circle ball collision physics.

Cheaper Collision Math

We reduced arithmetic inside of our “updateBall()” function via the following

- Compare the squared distance for 100 pixels² instead of the distance to 10 pickles. The bouncing
- A touching ball gets one set of “1 divided by squareroot”
- The velocity is divided by the substep count once, not on every subset
- A ball below the bottommost row skips subsets and peg lookup and moves.

```
fix15 dist2 = multfix15(dx,dx) + multfix15(dy,dy) ;
if ((dist2 < COLLIDE_DIST2) && (dist2 > 0)) {
    float inv_dist = 1.0f / sqrtf(fix2float15(dist2)) ;
    fix15 normal_x = (fix15)(dx * inv_dist) ;
    fix15 normal_y = (fix15)(dy * inv_dist) ;
```

Multicore Work Distribution

We split the animation work across core 0 and 1. Each frame has a 16,667 μ s deadline at 60 fps, and we want to keep both cores busy for the whole frame with as little synchronization as possible. This roughly doubled the compute available per frame.

Core 0	Core 1
Waits for vsync, wakes core 1	Waits for core0's signal
Clears top half of screen (row 0-239)	Clears bottom half of screen (row 240-479)
Polls push button, spawns new balls, updates split	Draws histogram and bin counts
Wait until both halves are clear	Wait until both halves are clear
Updates and draws balls [0, split)	Draws pegs, updates and draws balls [split, N)
Draws on-screen text, wait for core1, check deadline flag	Signals core 0 that its done

Table 4: Work Split Across Core 0 and Core 1

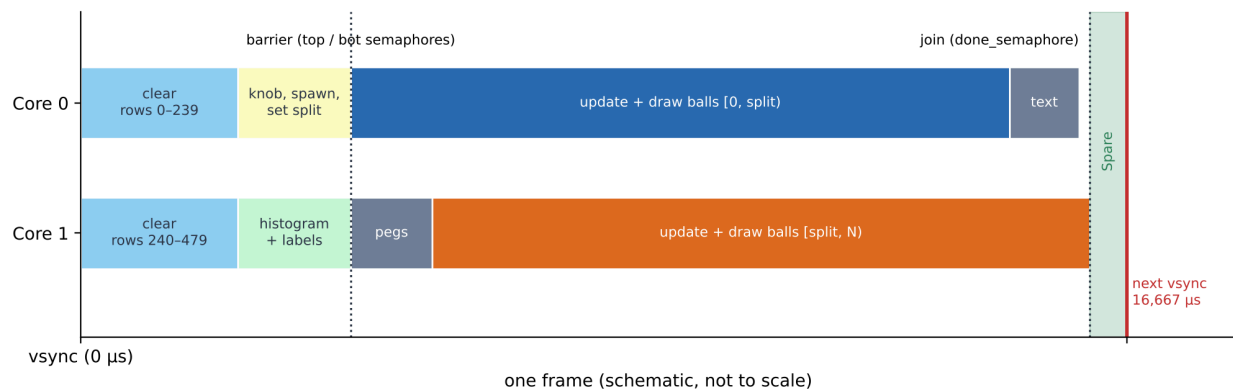


Figure 17: Visualization of Work Distribution Across Cores

Our first idea was a pipeline where one core does the physics and the other updates the screen, but that causes imbalancing between cores since drawing is the biggest bottleneck for speedup. Instead, for a reasonable load balancing across the two cores, we dynamically split the entire set of balls into 2 sets on every frame compute, with the variable *split* indicating the splitting point. Core 0 owns balls [0, split) and core 1 owns [split, N), and each core updates and draws its own balls back to back using the function *updateAndDrawBall()*. There's no lock or synchronization needed because no ball will be touched by both cores.

Where to put that split is the key to parallelizing both cores' load. A ball among the pegs runs collision substeps and a peg lookup every frame. As balls fall, the work inside any fixed range of indices keeps

changing, so a fixed 50/50 split never stays balanced for long. Since a frame only finishes when the slower core does, we let core 0 adjust the split every frame based on the previous frame's finish times:

```
int diff = (int)core1_us - (int)core0_us ;
if      (diff >  SPLIT_DEADBAND) split_pm = MIN(split_pm + 1, 1000) ;
else if (diff < -SPLIT_DEADBAND) split_pm = MAX(split_pm - 1, 0) ;
split = (num_balls * split_pm) / 1000 ;
```

Core 0's share *split_pm* is stored in thousandths and moves by 0.1% per frame toward whichever core finishes first. That's about 6% per second at 60 fps, fast enough to follow the balls through the board while staying stable. A 50 us deadband keeps it from reacting to normal timing noise. Because the share is a fraction rather than a ball count, it scales automatically when the user changes the number of balls. It starts at 60% on core 0, since core 1 also draws the pegs and the histogram.

In the one core design, we have a bin counter to obtain the histogram. To prevent the same bin counter which will lose counts when both cores bin a ball at the same time, we give each core its own counters. Each core only writes to its own row *bins_core[get_core_num()][bin]*, and the histogram adds the two rows when it draws them.

We synchronize the cores only at frame boundaries, so the ball loops run independently. Four semaphores, each initialized as a one shot signal, give every frame a fork-barrier-join structure. Core 0 starts core 1 with *draw_semaphore* after vsync. After clearing, the cores meet at the barrier:

```
// core 0                      // core 1
PT_SEM_SDK_SIGNAL(pt, &top_semaphore) ;    PT_SEM_SDK_SIGNAL(pt, &bot_semaphore) ;
PT_SEM_SDK_WAIT(pt, &bot_semaphore) ;      PT_SEM_SDK_WAIT(pt, &top_semaphore) ;
```

A ball can be anywhere on screen, so neither core draws until both halves are clear. Otherwise one core's clear would erase the other core's balls. At the end of the frame, core 0 waits on *done_semaphore* so it can't clear the next frame while core 1 is still drawing. We noticed the balls will flicker before we added this fix.

After we added the barrier, we made all shared states update before the barrier, like variables *num_balls*, *split*, newly spawned balls, and the histogram counters. Core 0 can change them only when core 1 is clearing or waiting at the barrier:

```
int target = rot_counter ;           // read the ISR's value once
for (int i = num_balls; i < target; i++) spawnBall(&balls[i]) ;
num_balls = target ;
split = (num_balls * split_pm) / 1000 ;
PT_SEM_SDK_SIGNAL(pt, &top_semaphore) ; // core 1 may now touch balls
```

Reading *rot_counter* once protects against the encoder ISR changing it mid-frame. The histogram reset happens even earlier, before core 1 is woken. As a result, core 1 always sees final values and never updates a ball that is being respawned.

The frame buffer is 640x480x4 bits = 153,600 bytes. Now core 0 wakes core 1 first, and each core clears 76,800 bytes in parallel, which halves the clear time:

```
clearRegion(0, CLEAR_SPLIT_Y, BLACK) ; // core 0: rows 0-239
clearLowFrame(CLEAR_SPLIT_Y, BLACK) ;  // core 1: rows 240-479
```

The histogram lies entirely below row 400, so core 1 also draws it after its own clear.

Memory Optimizations (Shrinking balls from 24 to 12 Bytes)

After overclocking to 300 MHz, RAM became the limit instead of the CPU. We had 3,150 μ s to spare at 10,000 balls but couldn't add more. The two frame buffers take 307.2 KB of the 520 KB of SRAM, and at 24 bytes per ball the linker overflowed above about 8,000 balls. Thus to optimize our memory usage, we shrank the ball struct in three steps:

Version	Layout	Size	Max balls
Original	fix15 x, y, vx, vy; int last_peg; int binned	24 B	~8,000
Step 1	last_peg $\rightarrow$ int16_t, binned $\rightarrow$ uint8_t	20 B	10,300
Attempt	velocity as int16 with 8 fractional bits	16 B	histogram broke
Step 2	flags packed into spare bits of a full-precision vx	16 B	12,500
Final	fix15 position with flags in the low 5 bits; fix10 velocity	12 B	~17,000

Table 5: Ball Struct Optimization for RAM

The reverted attempt was the most useful failure. The spawn velocity is the board's only source of randomness, and at 8 fractional bits the spawn range held just 128 distinct values, so balls could take only 128 different paths. In simulation this produced false histogram spikes and 144 stuck balls out of 20,000. Those were spawns that rounded to $vx = 0$ and bounced on the top peg forever.

The final layout (Figure 3) keeps full fix15 precision for position. It uses the 5 spare low bits of each position word for the 9 flag bits, $meta = (last_peg + 1) \ll 1 \mid binned$, and stores velocity in fix10, which covers ± 32 px/frame (the fastest ball we saw reached 16.4). To keep enough distinct paths, spawnBall() forces vx to be odd so it can never be zero, and adds ± 0.5 px of position jitter:

```
fix15 vx = vel2fix15(((rand() & 0x1FF) - 0x100) | 1); // odd: never 0
```

At 12 bytes, MAX_BALLS = 17,000 uses 204 KB, and the CPU became the limit again.

Code Execution Optimization (Running code from the SRAM)

With the 12-byte struct, RAM was no longer the problem, but at 350 MHz the CPU hit its limit at 13,000 balls with 0 μ s of spare time. The cause wasn't how much work each ball needed, but where that work was being run from. By default the RP2350 executes code straight out of the external QSPI flash chip, through an execute-in-place (XIP) cache. That cache is only 16 KB, and both cores share it. Since both cores run the same physics and drawing code on their own share of the balls every frame, they compete for the same cache, and every miss stalls a core while it waits for instructions from flash.

Looking at the disassembly showed a second, sneakier cost. GCC recognized the byte-fill loop in *drawSpan()* as a *memset* and replaced it with a library call. That function lives in flash, so drawing a single ball branched out of our code up to 10 times per frame, each time paying a full function call just to write 1–4 bytes.

To fix both, we moved every function that runs once per ball into on-chip SRAM, using the Pico SDK's section attributes:

Change	Implementation	Purpose
Ball loop in SRAM	New <i>updateAndDrawRange(from, to)</i> . Marked <code>__no_inline_not_in_flash_func</code>	Both cores call it for their share of the balls; the inline drawing helpers (<i>drawBall</i> , <i>drawRingRow</i> , <i>drawSpan</i>) are compiled into it, so they run from SRAM too
Physics in SRAM	<i>updateBall</i> , <i>nearestPeg</i> , <i>binBall</i> , <i>playPegSound</i> . Marked <code>__not_in_flash_func</code>	Collision code runs from SRAM; the compiler inlined the last three into <i>updateBall</i>
No library fill calls	<i>drawSpan()</i> writes its bytes with up to four direct stores	Removes the <i>memset</i> calls into flash

Table 6: Moving the Per-Ball Code from Flash to SRAM

The obvious fix didn't work at first. With only `__not_in_flash_func` on the loop, *-Ofast* inlined it right back into the two animation threads, which still run from flash, so none of the hot code actually moved. We only caught this by checking function addresses in the compiled *.elf*. Adding *no_inline* forces the loop to stay its own function in SRAM:

```
void __no_inline_not_in_flash_func(updateAndDrawRange)(int from, int to)
{
    for (int i = from; i < to; i++) updateAndDrawBall(&balls[i]) ;
}
```

Each core then calls it for its own share of the balls:

```
updateAndDrawRange(0, split) ; // core 0
updateAndDrawRange(split, num_balls) ; // core 1
```

For *drawSpan()*, we know a ball row is at most $2 \times BALL_RADIUS = 8$ px, which is 4 bytes, so four guarded stores always cover it:

```
int n = w >> 1 ;
if (n > 0) p[0] = both ;
if (n > 1) p[1] = both ;
if (n > 2) p[2] = both ;
if (n > 3) p[3] = both ;
```

This is also why *BALL_RADIUS* is now capped at 4. After rebuilding, we checked the *.elf* again:

- *updateAndDrawRange* (1.2 KB) and *updateBall* (0.9 KB) both sit in SRAM, at *0x2000xxxx* addresses.
- Only two calls from this path still go out to flash:
- *drawCircle()*, used only for balls touching the screen edge;
- *rand()*, used only when a ball respawns, a few dozen times per frame.
- The whole change costs about 2 KB of RAM.

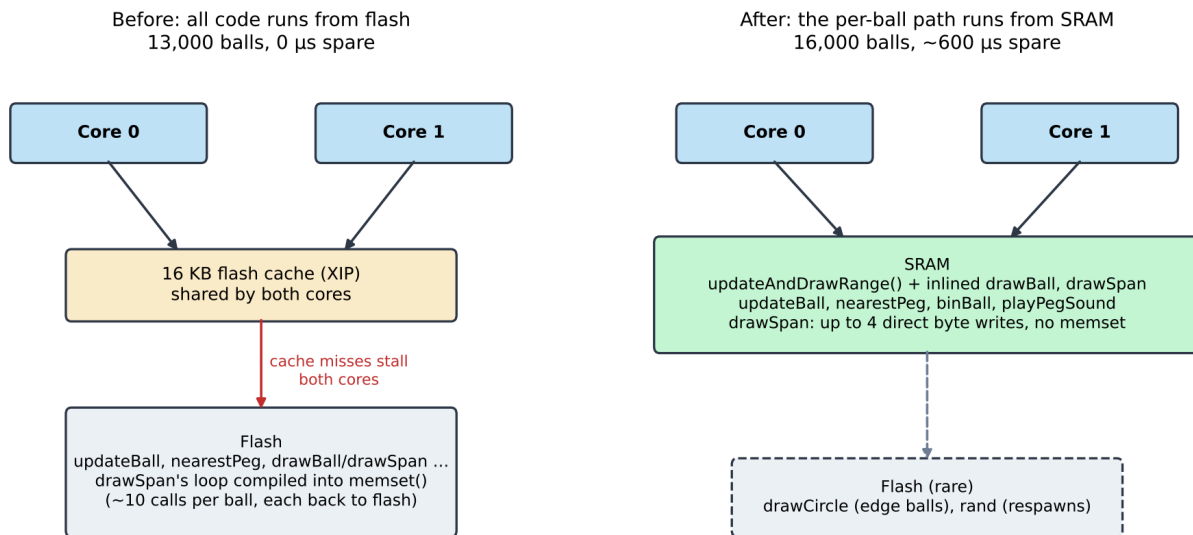


Figure 18: Before and after, moving ball functions to on-chip SRAM

We expected a 5–20% gain. On the board, the limit went from 13,000 balls at 0 μ s spare to 16,000 balls with about 600 μ s spare, about 25% more balls from code that does exactly the same math. That gain is the combined effect of the SRAM placement and removing the *memset* calls; we didn't measure the two separately.

Histogram (Results / Lab 2 Analysis Questions)

To validate the idea that the result of balls falling down a Galton board produces the shape of a gaussian distribution, we implemented a histogram.

Before diving into the histograms, we must discuss the following: Our physical model of the Galton Board differs from the idealized model. In order for the Central Limit Theorem to hold, all random numbers must be **independent, identically distributed**, and have **finite variance**.

1. Which of these assumptions hold for our physical Galton Board model, and which do not?

→ Independence: Does NOT hold!

- Only randomness is the spawning condition. Thereafter, motion becomes deterministic! Consecutive steps are correlated as the way in which the balls goes at any given row depend on position and velocity from the previous row

→ Identically Distributed: Does NOT hold!

- Odds of left and right depend on how fast the ball hits!

→ Finite Variance: DOES hold!

- At all steps, balls are moved by a bounded distance, and positions and velocities are bounded, therefore every step has finite variance.

2. Why do we still see a distribution that approaches a Gaussian?

We still see a distribution that approaches a Gaussian distribution as some generation of CLT relaxes these assumptions. In other words, we still see a bell curve since the central limit theorem still works. Steps don't have to be exactly alike AS LONG AS no single step matters more than another, they also don't have to be independent. Each step will remember the last step, not the whole past. Our board has both of these behaviors: Every bounds blows up tiny differences, then after a few rows it "forgot" which way it went earlier. Each of the 16 rows also acts like new coin flips: adding up 16 nearly random left and right steps will provide a bell curve!

This stops working when the ball becomes more bouncy. Ball ends up skipping pegs so single step divides most of where balls land, making the histogram not look like a bell curve anymore!

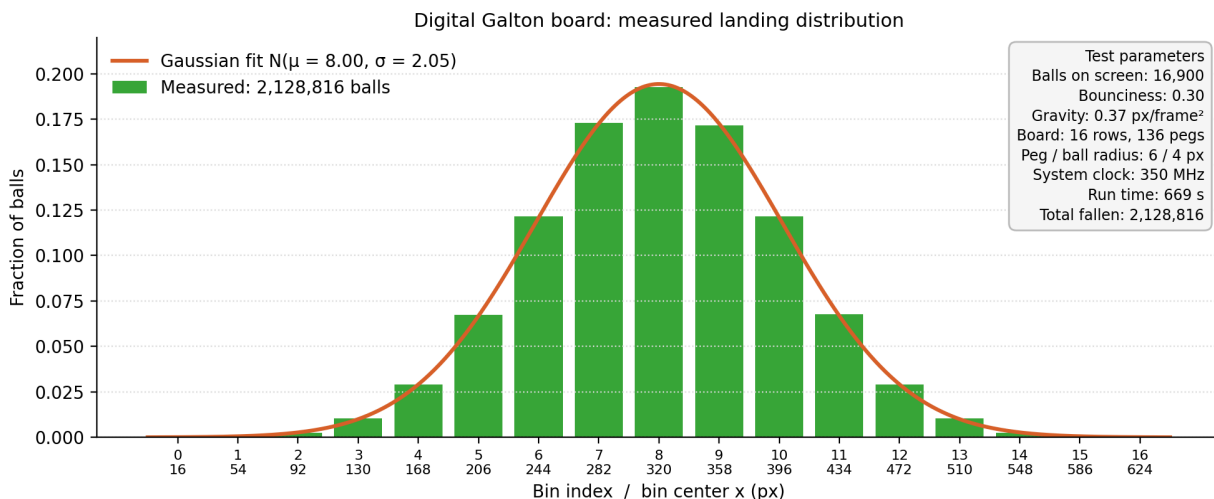


Figure 19: Actual Galton Board Histogram Distribution

For our 16-row Galton Board, for the Gaussian which approximates the resulting distribution, the:

→ **Mean is 8.00**

→ **Standard deviation is 2.05**

For an ideal board, the bin index is Binomial(16, 1/2):

$$\mu = n/2 = 8, \quad \sigma^2 = n/4 = 4, \quad \sigma = 2 \text{ bins}$$

Below are the final results compared to the theoretical results as discussion of the gaussian distribution in our "Mathematical Background" section:

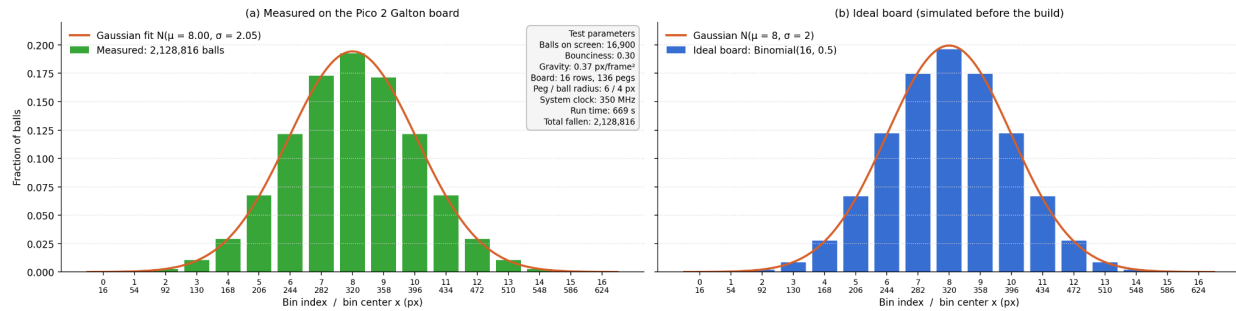


Figure 20: Actual vs Ideal Galton Board Histogram Distribution

As we see, the Gaussian looks nearly identical to that of the theoretical Gaussian, showing how awesome our Galton Board came out!

The Gaussian approximates the distribution under the Galton board better with more rows in the board. Why?

→ With more bins, become a fine grained distribution, providing more steps of less error to provide a smoother Gaussian distribution

Think about it: Every new row adds an extra random step to the sum. The central limit theorem is concerned about the limit of numerous steps, where the error of normal approximation shrinks as:

$$1/\sqrt{n} \text{ (the Berry-Esseen bound)}$$

If there is less error, this means the Galton Board gets better! We would no longer have a steeper curve, and have each bar concerned with a smaller fraction of the curve's width.

Application Level

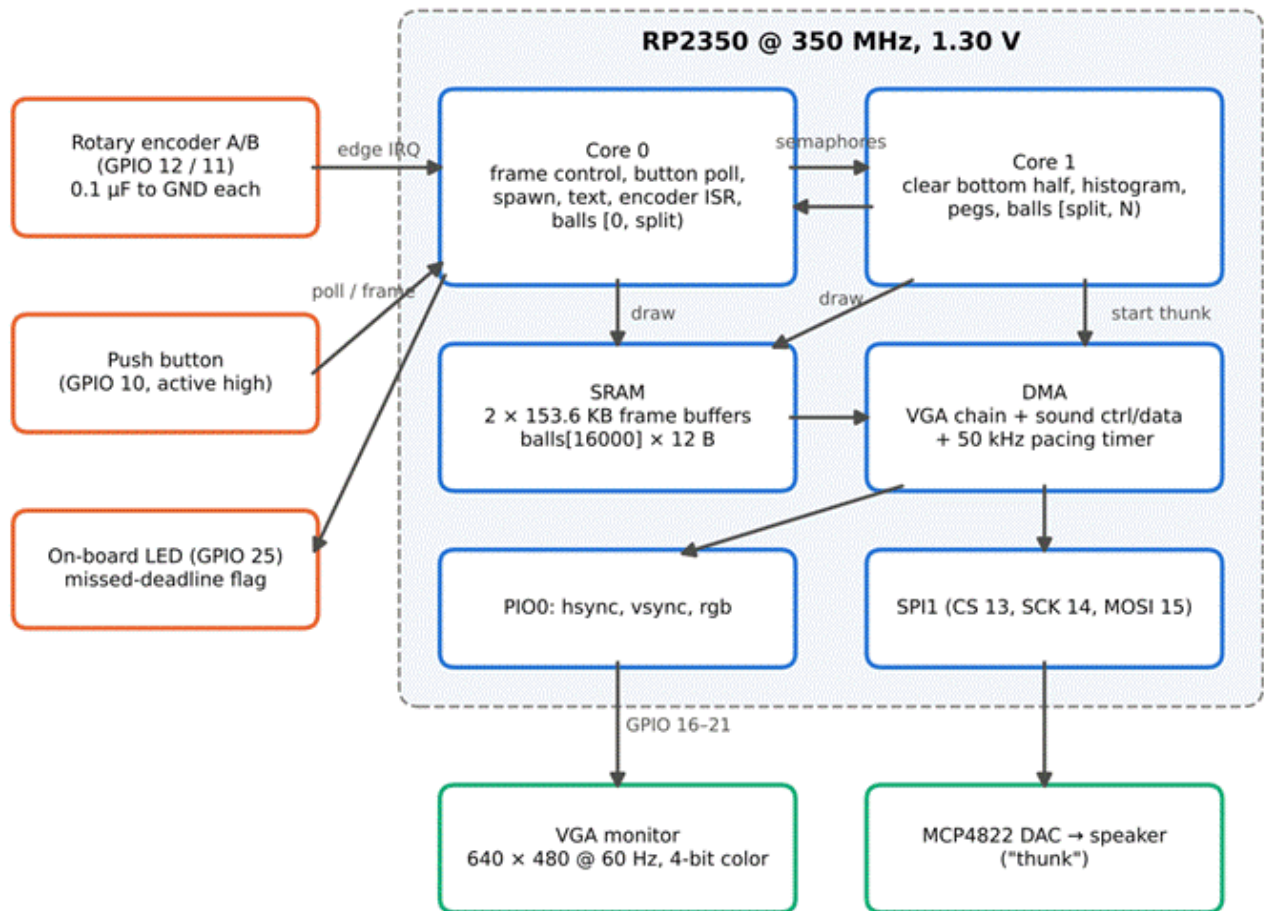


Figure 21: System Level Block Diagram

Here is a picture of our circuit breadboarded:

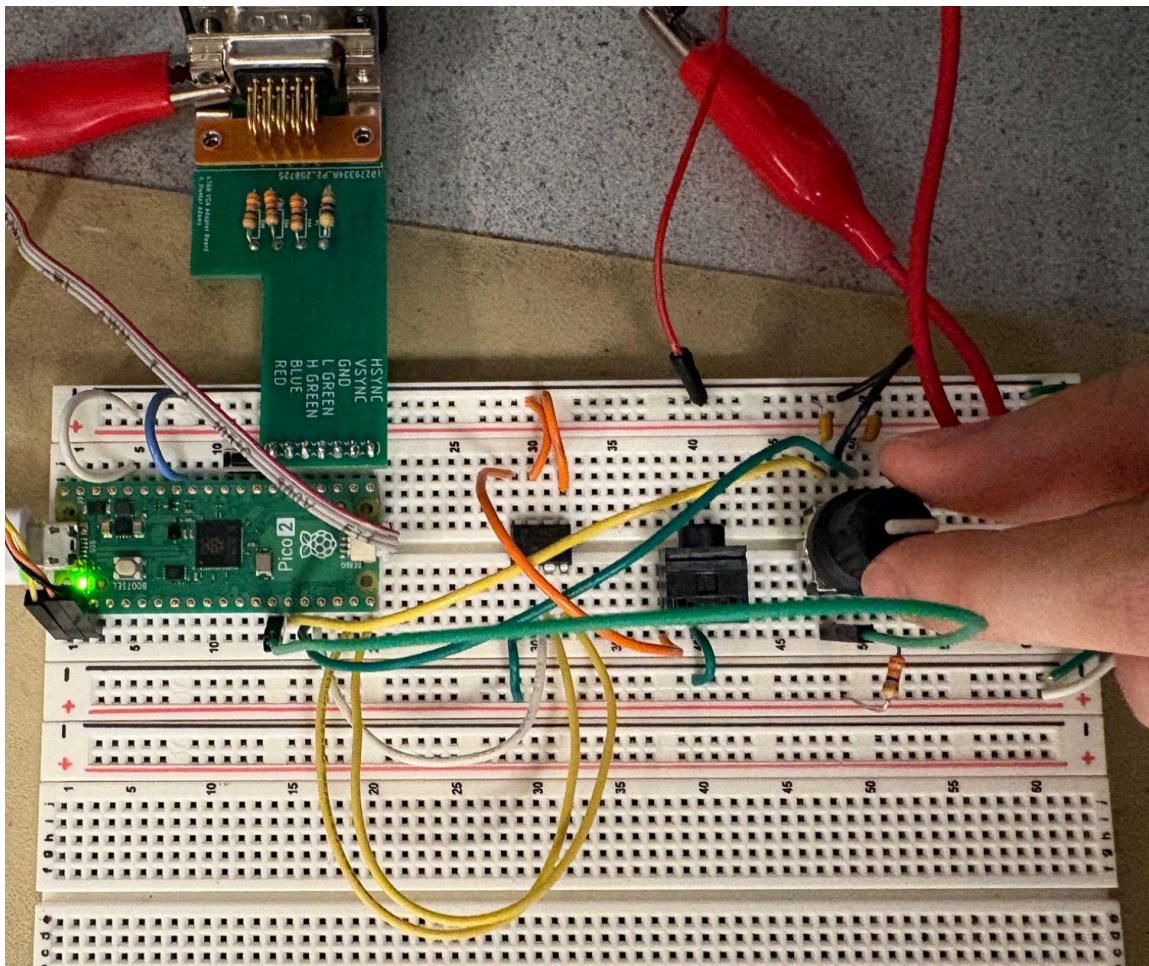


Figure 22: Breadboard Hardware Setup

Conclusion:

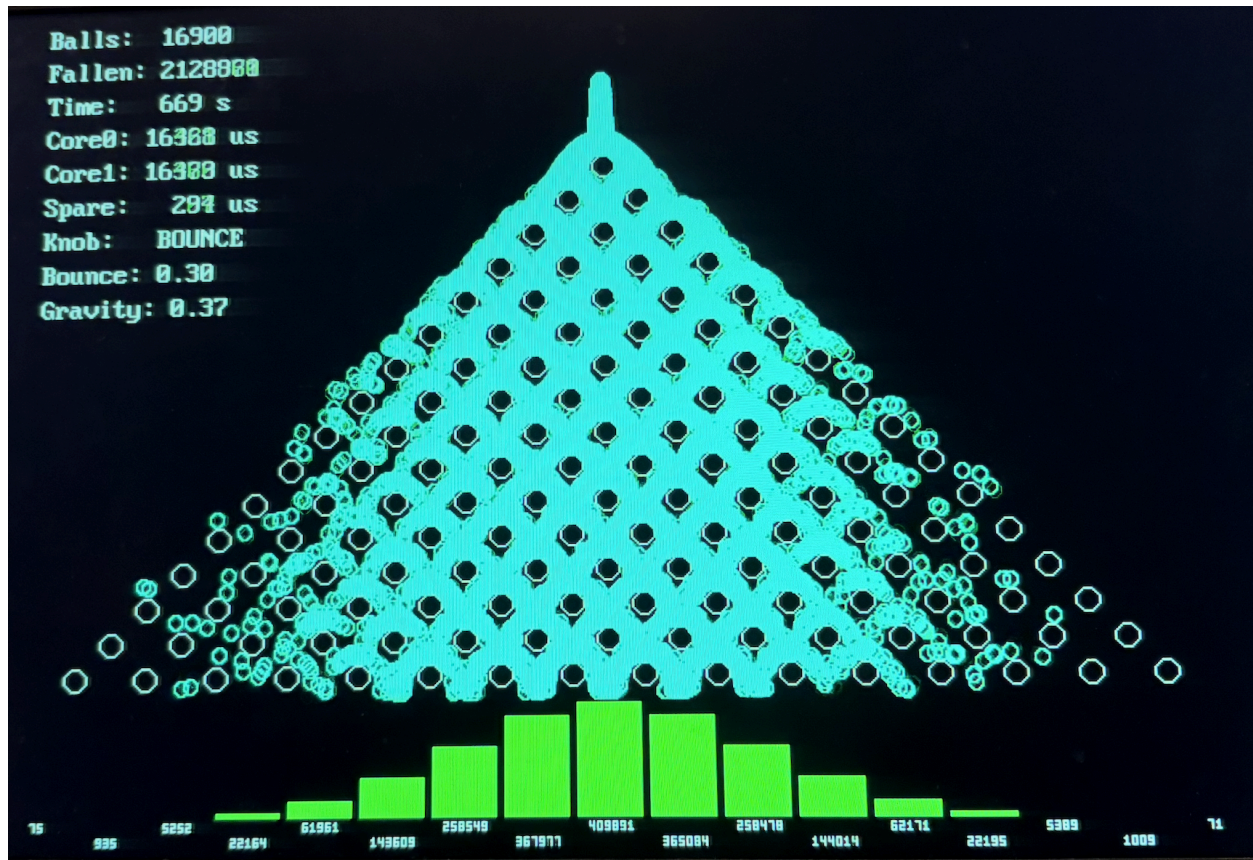


Figure 23: Final Galton Board Display

We could have optimized our system even more, but we were time bound! (Ran out of lab time before lab3 came around.) In this lab, we first completed a baseline design:

- Working digital Galton board
- Collision Physics
- DMA Sound
- Rotary Encoder Interface
- Normalized Histogram

We learned how to interface new hardware, debounce and decode via software to control a new graphics software, a new feat in this course thus far!

Optimizations were then completed afterwards:

- O(1) nearest-pnf lookup
- Split balls across both cores
- Precomputed Ball Sprite + Float literals
- Cheaper collision math
- Parallel clear + auto split
- Overlock
- Ball struct decrease of bytes

- Hot code in SRAM

We learned how to squeeze out the most of our limited resources. An emerging skill in today's world, especially as memory prices ramp up, we need to relearn how to write efficient code on limited memory, no more bloated software wasting resources or leaving resources unused.

Work Distribution:

Amber, Ivan, and Zarif worked together on all parts of the lab. We frequently met outside of our lab sections, and most of our work was done in person in the lab. Outside of the lab, we communicated frequently to ensure that we were all aware of the progress that had been made and what the immediate next steps were. The contents of this report were written by us.

Appendix A: Heavily Commented Code

```

/**
 * ECE 4760 Lab 2 -- Digital Galton Board (RP2350 / Pico 2)
 * Built on Hunter Adams' (vha3@cornell.edu) VGA animation demo.
 *
 * AUTHORS
 * - Amber Li
 * - Ivan Huang
 * - Zarif Karim
 * Date: October 15, 2026
 *
 * Balls drop onto a 16-row peg board and land in a histogram. A rotary
 * encoder sets the ball count, bounciness, or gravity; each peg strike plays
 * a short DMA-driven sound. Both cores update and draw the balls at 60 fps.
 *
 * HARDWARE CONNECTIONS
 * - GPIO 16 ---> VGA Hsync
 * - GPIO 17 ---> VGA Vsync
 * - GPIO 18 ---> VGA Green lo-bit --> 470 ohm resistor --> VGA_Green
 * - GPIO 19 ---> VGA Green hi_bit --> 330 ohm resistor --> VGA_Green
 * - GPIO 20 ---> 330 ohm resistor ---> VGA-Blue
 * - GPIO 21 ---> 330 ohm resistor ---> VGA-Red
 * - RP2350 GND ---> VGA-GND
 *
 * Rotary encoder
 * GPIO 12 green left side. A
 * GPIO 11 yellow right side B
 * GPIO 10 push button (active high)
 *
 * MCP4822 DAC (spi1)
 * GPIO 13 ---> CS
 * GPIO 14 ---> SCK
 * GPIO 15 ---> MOSI (SDI)
 *
 * GPIO 25 on-board LED: lit when a frame misses the 60 fps deadline
 *
 * RESOURCES USED
 * - PIO state machines 0, 1, and 2 on PIO instance 0
 * - DMA channels (VGA, plus 2 for the peg sound) and one DMA pacing timer
 * - 2 x 153.6 kBytes of RAM (double-buffered pixel data)
 *
 * OPTIMIZATION LOG -- max balls at 60 fps (each step tagged [OPT n] in the code)
 * OPT 1 O(1) nearest-peg lookup ..... 136 peg tests/substep -> 1 (enabled multi-ball)
 * OPT 2 Split balls across both cores .. ~2x compute (not measured alone)
 * OPT 3 Precomputed ball sprite ..... 150 MHz: 5,000 balls
 * OPT 4 Cheaper collision math ..... 150 MHz: 5,200 balls
 * OPT 5 Parallel clear + auto split .... shipped with OPT 6 (not measured alone)
 * OPT 6 Overclock 150 -> 300 -> 350 MHz 300 MHz: 3,150 us spare at 10k; 350 MHz: 13,000 (0 us spare)
 * OPT 7 Ball struct 24 -> 12 bytes ..... RAM ceiling 8k -> 10.3k -> 12.9k -> ~17k
 * OPT 8 Hot code in SRAM ..... 350 MHz: 16,000 balls with ~600 us spare
 *
 * AI-ASSISTED CODE -- PROMPT LOG
 * Claude Code (and later Codex) helped write parts of this file. The prompts
 * and responses are recorded in SESSION_LOG-2.md; each tagged region below
 * carries a [PROMPT #n] comment that matches entry #n in that log.
 * #1-#10 rotary encoder: pull-ups, 4-state quadrature ISR, rot_table, accumulator
 * #11 one ball + one peg: spawnBall(), updateBall(), reflection math, GPIO setup

```

```

* #12    COLLIDE_DIST / TELEPORT_DIST and pointer-based spawn/update signatures
* #13    respawn at the bottom of the SCREEN (removed the phantom inner-box bounce)
* #16    coefficient-of-restitution attempt (later replaced by the bounciness scale)
* #18    histogram + Fallen count reset when a knob parameter changes
* #19-#20 spawn velocity forced odd so a ball can't sit on the top peg
* #21    bin-count labels under the histogram bars
* #22    stop drawing balls once they pass the bottom peg row
* #26    this commenting pass (Authors block, tags, extra comments)
* The OPT 1-8 optimizations are documented by the [OPT n] blocks and in
* Lab2_Galton_Notes.md; they are not in the session log.
*/

// Include the VGA graphics library
#include "VGA/vga16_graphics_v3.h"    // VGA driver: drawPixel, fillRect, drawCircle, text, buffers
// Include standard libraries
#include <stdio.h>                     // sprintf / snprintf for on-screen text
#include <stdint.h>                    // fixed-width integer types (int16_t, uint8_t, ...)
#include <stdlib.h>                    // rand(), abs()
#include <math.h>                      // sinf() for the sound table, sqrtf() for collisions
#include <string.h>                    // memset() to clear the histogram
// Include Pico libraries
#include "pico/stdlib.h"              // Pico SDK: GPIO, timing, busy_wait
#include "pico/divider.h"              // hardware divider (div_s64s64 used by divfix)
#include "pico/multicore.h"            // launch and talk to core 1
#include "pico/sync.h"                 // semaphores for the core 0 / core 1 fork-join
// Include hardware libraries
#include "hardware/pio.h"              // PIO (VGA signal generation)
#include "hardware/dma.h"              // DMA (VGA frame buffer + peg sound)
#include "hardware/clocks.h"           // clock_configure / clock_get_hz (overclock)
#include "hardware/pll.h"              // pll_init (350 MHz system PLL)
#include "hardware/spi.h"              // SPI to the MCP4822 DAC
#include "hardware/vreg.h"             // core voltage regulator (needed to overclock)
// Include protothreads
#include "pt_cornell_rp2040_v1_4.h"    // pt = protothreads (cooperative scheduler)

// === the fixed point macros =====
// fix15 = signed Q15: 1 sign bit, 16 integer bits, 15 fractional bits, so
// 1.0 = 32768. Integer math only, because the M33 FPU is slow for this much work.
// A fix15 * fix15 product is 30 fractional bits, so multfix15 widens to 64 bits
// and shifts back down by 15.
typedef signed int fix15 ;
#define multfix15(a,b) (((fix15)((((signed long long)(a))*((signed long long)(b))>>15)))
#define float2fix15(a) ((fix15)((a)*32768.0f)) // [OPT 3] float, not double: the M33 FPU is single-precision only
#define fix2float15(a) ((float)(a)/32768.0f)
#define absfix15(a) abs(a)
#define int2fix15(a) ((fix15)(a << 15))      // integer -> fix15 (exact)
#define fix2int15(a) ((int)(a >> 15))        // fix15 -> integer (arithmetic shift, floors toward -inf)
#define char2fix15(a) (fix15)((((fix15)(a)) << 15))
// fix15 divide: pre-shift the numerator by 15 in 64 bits, hardware 64-bit divider
#define divfix(a,b) (fix15)(div_s64s64( (((signed long long)(a)) << 15), (((signed long long)(b))))

// unused (left over from the original demo; frame timing comes from the VGA driver)
#define FRAME_RATE 33000

// =====
// === ROTARY ENCODER INTERFACE =====
// =====
#define ROT_A 12
#define ROT_B 11

```

```

#define ROT_SW 10    // push button: reads HIGH while pressed

// Knob modes; the push button steps BALLS -> BOUNCE -> GRAVITY -> BALLS
#define ROT_MODE_BALLS    0
#define ROT_MODE_BOUNCE  1
#define ROT_MODE_GRAVITY  2
#define ROT_NUM_MODES    3
volatile int rot_mode = ROT_MODE_BALLS ;    // written by pollButton() (core 0), read by rot_click() in the ISR

#define MIN_BALLS    1
#define MAX_BALLS    17000    // [OPT 8] measured limit: ~600 us spare at 350 MHz (RAM would allow ~17k)
#define INIT_BALLS    16500
#define BALL_STEP    100
volatile int rot_counter = INIT_BALLS ;    // = number of balls to animate

// Knob parameters are kept in hundredths for the knob/display, mirrored as fix15 for the physics
#define MIN_BOUNCE    0
#define MAX_BOUNCE    100
#define INIT_BOUNCE    30
#define BOUNCE_STEP    5
volatile int    bounce_pct = INIT_BOUNCE ;
volatile fix15 bounciness = (INIT_BOUNCE << 15) / 100 ;

// Gravity (px/frame^2). Capped at 1.00 so a full-height fall (~29 px/frame)
// stays inside ball_t's +/-32 px/frame velocity range [OPT 7]
#define MIN_GRAV    -5
#define MAX_GRAV    100
#define INIT_GRAV    37
#define GRAV_STEP    5
volatile int    grav_pct = INIT_GRAV ;
volatile fix15 gravity    = (INIT_GRAV << 15) / 100 ;

// [PROMPT #18] "can you make it so the histogram and total fallen balls resets
// when any parameter is changed using the knob?"
// Set when a knob click changes a parameter; core 0 then clears the histogram
// and the fallen count at the start of the next frame
volatile int stats_reset = 0 ;

// [PROMPT #7, #10] "i feel the brush a provide one incrment and brush b provide
// another... maybe you do have to read a then b and keep track of the sequence
// of events" -> full 4-state quadrature tracking instead of A-edge-only.
// States are (A<<1)|B: 0=00, 1=01, 2=10, 3=11. A detent rests at 3 (both
// pull-ups high); one click walks 3 -> 2 -> 0 -> 1 -> 3 (CW) or the reverse (CCW).
static volatile uint8_t rot_prev_state = 3 ; // (A<<1)|B ; 3 = rest (both high)
static volatile int8_t  rot_accum = 0 ;    // steps taken since last rest

// Gray-code step table, index = (prev<<2)|curr: +1 CW, -1 CCW, 0 = no step or bounce
// (0 also covers illegal jumps that skip a state, which are noise)
static const int8_t rot_table[16] = {
/* prev=0(00) */ 0, -1, +1, 0,
/* prev=1(01) */ +1, 0, 0, -1,
/* prev=2(10) */ -1, 0, 0, +1,
/* prev=3(11) */ 0, +1, -1, 0
};

// One knob detent: adjust the current mode's value (clamped); request a
// stats reset only if the value actually changed
static void rot_click(int dir)    // dir = +1 (CW) or -1 (CCW), from rot_ISR
{

```

```

int changed ; // 1 only if clamping left the value different
if (rot_mode == ROT_MODE_BALLS) {
    int v = MIN(MAX(rot_counter + dir * BALL_STEP, MIN_BALLS), MAX_BALLS) ;
    changed = (v != rot_counter) ;
    rot_counter = v ;
} else if (rot_mode == ROT_MODE_BOUNCE) {
    int v = MIN(MAX(bounce_pct + dir * BOUNCE_STEP, MIN_BOUNCE), MAX_BOUNCE) ;
    changed = (v != bounce_pct) ;
    bounce_pct = v ;
    bounciness = (v << 15) / 100 ; // hundredths -> fix15 (e.g. 30 -> 0.30)
} else {
    int v = MIN(MAX(grav_pct + dir * GRAV_STEP, MIN_GRAV), MAX_GRAV) ;
    changed = (v != grav_pct) ;
    grav_pct = v ;
    gravity = (v << 15) / 100 ;
}
if (changed) stats_reset = 1 ;
}

// Push button, polled once per frame (slower than contact bounce, so no debounce).
// RP2350 erratum E9: the input buffer is enabled only while sampling, or the
// pulled-down pin can latch high after release.
static void pollButton(void)
{
    static int sw_prev = 0 ;
    gpio_set_input_enabled(ROT_SW, true) ;
    busy_wait_us(1) ; // let the input synchronizer settle
    int sw = gpio_get(ROT_SW) ;
    gpio_set_input_enabled(ROT_SW, false) ;
    if (sw && !sw_prev) {
        rot_mode = (rot_mode + 1) % ROT_NUM_MODES ; // BALLS -> BOUNCE -> GRAVITY -> BALLS
        rot_accum = 0 ; // don't carry a partial turn into the new mode
    }
    sw_prev = sw ;
}

// [PROMPT #1-#10] "Read the rotary encoder page and write a software interface
// to the rotary encoder..." (final form from #10).
// Fires on every edge of A or B. A click counts only after 4 steps back at
// rest, so contact bounce (+1 then -1) cancels out.
// gpio / events are the SDK callback arguments; both pins share this one handler,
// so they are unused (we re-read both pins instead).
void rot_ISR(uint gpio, uint32_t events)
{
    uint8_t curr_state = (gpio_get(ROT_A) << 1) | gpio_get(ROT_B) ;
    rot_accum += rot_table[(rot_prev_state << 2) | curr_state] ;
    rot_prev_state = curr_state ;

    if (curr_state == 3) { // landed back at rest (a full click, or none)
        if (rot_accum >= 4) rot_click(+1) ; // CW
        else if (rot_accum <= -4) rot_click(-1) ; // CCW
        rot_accum = 0 ; // discard partial turns / bounce
    }
}

// =====
// === GALTON BOARD =====
// =====
// gravity and bounciness (set by the knob) are defined with the rotary encoder above

```

```

// [PROMPT #11] "Use the default parameters... peg_radius = 6, gravity = 0.37,
// bounciness=0.5, ball_radius = 4... vertical separation = 19, horizontal
// separation = 38." (gravity and bounciness are now knob-controlled; see above)
#define BALL_RADIUS 4 // max 4: drawSpan() writes at most 4 bytes per span [OPT 8]
#define PEG_RADIUS 6
#define PEG_VERT_SEP 19 // px between rows
#define PEG_HORZ_SEP 38 // px between pegs in a row (also the histogram bin width)
#define SCREEN_W 640
#define SCREEN_H 480

// [PROMPT #12] build errors: COLLIDE_DIST / TELEPORT_DIST were undeclared
// collision distances in fix15 (center-to-center)
#define COLLIDE_DIST int2fix15(BALL_RADIUS + PEG_RADIUS) // touching: ball edge meets peg edge
#define TELEPORT_DIST int2fix15(BALL_RADIUS + PEG_RADIUS + 1) // 1 px clear of contact, so no re-collision
// [OPT 4] squared collision distance, so the contact test needs no sqrt
#define COLLIDE_DIST2 int2fix15((BALL_RADIUS + PEG_RADIUS) * (BALL_RADIUS + PEG_RADIUS))

// Board layout: row r (0..15) has r+1 pegs, centered on BOARD_TOP_X.
// Bottom row sits at BOARD_TOP_Y + 15*19 = 385, leaving room for the histogram.
#define NUM_ROWS 16
#define NUM_PEGS (NUM_ROWS * (NUM_ROWS + 1) / 2) // 136
#define BOARD_TOP_X (SCREEN_W / 2)
#define BOARD_TOP_Y 100

// Peg centers, indexed row by row: peg = row*(row+1)/2 + col
fix15 peg_x[NUM_PEGS] ;
fix15 peg_y[NUM_PEGS] ;

// Fill in the peg table once at startup
void initPegs()
{
    int peg = 0 ;
    for (int row = 0; row < NUM_ROWS; row++) {
        for (int col = 0; col <= row; col++) {
            // leftmost peg of each row is half a row-width left of center
            peg_x[peg] = int2fix15(BOARD_TOP_X - row * (PEG_HORZ_SEP / 2) + col * PEG_HORZ_SEP) ;
            peg_y[peg] = int2fix15(BOARD_TOP_Y + row * PEG_VERT_SEP) ;
            peg++ ;
        }
    }
}

// =====
// [OPT 1] O(1) NEAREST-PEG LOOKUP
// Bottleneck : every ball tested all 136 pegs on every substep.
// Approach : rows are 19 px apart and contact needs < 10 px, so only the
// nearest peg can touch the ball. Compute its row and column
// directly from (x, y) with two divisions.
// Effect : collision search drops from 136 tests to 1 per substep;
// the step that made a multi-ball board possible.
// Expected / measured : ~100x less collision work; not measured alone
// (it predates the multi-ball code).
// =====
// Returns the index of the peg nearest (x, y), or -1 if none is in range.
int __not_in_flash_func(nearestPeg)(fix15 x, fix15 y)
{
    int yi = fix2int15(y) - BOARD_TOP_Y + (PEG_VERT_SEP / 2) ;
    if (yi < 0) return -1 ; // above the board
    int row = yi / PEG_VERT_SEP ;
    if (row >= NUM_ROWS) return -1 ; // below the board

```

```

int xi = fix2int15(x) - (BOARD_TOP_X - row * (PEG_HORZ_SEP / 2)) + (PEG_HORZ_SEP / 2) ;
if (xi < 0) return -1 ; // left of this row
int col = xi / PEG_HORZ_SEP ;
if (col > row) return -1 ; // right of this row

return row * (row + 1) / 2 + col ;
}

// =====
// [OPT 7] BALL STRUCT 24 -> 12 BYTES
// Bottleneck : RAM. The two VGA frame buffers take 307 KB of the 520 KB
// SRAM; at 24 bytes/ball the linker overflowed above ~8,000.
// Approach : shrink each ball without losing physics accuracy.
// 24 B -> 20 B: narrow last_peg / binned.
// 20 B -> 16 B: pack the flags into spare bits of vx.
// 16 B -> 12 B (below): position keeps full fix15 in bits
// 31..5 (+/-2048 px; balls reach ~1300 px at bounciness 1),
// the 9 flag bits ("meta" = (last_peg+1)<<1 | binned) live in
// the 5 low bits of xm and ym, and velocity is fix10
// (1/1024 px/frame, +/-32 px/frame; max seen ~17).
// Validation : an earlier fix8 velocity broke the histogram (only 128
// distinct paths; ~1/128 balls stuck on the top peg). This
// layout was checked in a PC simulation of this exact physics
// (20k drops at bounciness 0.3/0.5/0.6/1.0): 0 stuck balls and
// the same histogram as full fix15, given the two spawnBall()
// changes (odd vx, sub-pixel drop jitter).
// Expected / measured : RAM ceiling 8k (24 B) -> 10.3k (20 B) ->
// 12.9k (16 B) -> ~17k (12 B). Measured: 10,300 balls (20 B),
// 12,500 (16 B); at 12 B the CPU became the limit again.
// =====
typedef struct {
    int32_t xm, ym ; // position: fix15 in bits 31..5, meta flag bits in bits 4..0
    int16_t vx, vy ; // velocity: fix10 (1/1024 px per frame)
} ball_t ;
_Static_assert(sizeof(ball_t) == 12, "ball_t grew -- MAX_BALLS may no longer fit in RAM") ;

#define BALL_POS(m) ((fix15)((m) >> 5)) // arithmetic shift: exact
#define BALL_META(b) (((b)->xm & 31) | (((b)->ym & 31) << 5)) // gather the 9 flag bits from xm and ym
#define META_LAST_PEG(m) (((m) >> 1) - 1) // stored +1 so "no peg yet" (-1) is 0
#define META_BINNED(m) ((m) & 1) // 1 = already counted in the histogram
#define MAKE_META(peg, binned) (((peg) + 1) << 1) | (binned)
#define POS_LIMIT int2fix15(2047)
#define vel2fix15(v) ((fix15)(v) * 32) // fix10 -> fix15, exact

// [OPT 7] fix15 -> fix10: round to nearest (a plain >> biases balls left/up), saturate to int16
static inline int16_t fix2vel(fix15 a)
{
    a = (a + 16) >> 5 ;
    return (int16_t)MIN(MAX(a, -32767), 32767) ;
}

// [OPT 7] Pack a ball; positions clamped to +/-2047 px so the << 5 can't overflow
static inline void ballStore(ball_t* b, fix15 x, fix15 y, fix15 vx, fix15 vy, int meta)
{
    x = MIN(MAX(x, -POS_LIMIT), POS_LIMIT) ;
    y = MIN(MAX(y, -POS_LIMIT), POS_LIMIT) ;
    b->xm = (int32_t)((uint32_t)x << 5) | (meta & 31) ;
    b->ym = (int32_t)((uint32_t)y << 5) | (meta >> 5) ;

```

```

    b->vx = fix2vel(vx) ;
    b->vy = fix2vel(vy) ;
}

ball_t balls[MAX_BALLS] ;    // 17,000 x 12 B = 204 KB of SRAM
int num_balls = 0 ;          // balls currently animated: balls[0 .. num_balls-1]

// === HISTOGRAM ===
// 16 rows -> 17 landing spots: the 15 gaps between bottom-row pegs, plus one
// past each end. Bin k is centered PEG_HORZ_SEP*k right of BIN0_CENTER_X.
#define NUM_BINS      (NUM_ROWS + 1)
#define BOTTOM_ROW_Y   (BOARD_TOP_Y + (NUM_ROWS - 1) * PEG_VERT_SEP)    // 385
#define BOTTOM_LEFT_X  (BOARD_TOP_X - (NUM_ROWS - 1) * (PEG_HORZ_SEP / 2)) // 35
#define BIN0_CENTER_X  (BOTTOM_LEFT_X - PEG_HORZ_SEP / 2)
// A ball is counted once its top edge clears the bottom row of pegs
#define BIN_LINE_Y     int2fix15(BOTTOM_ROW_Y + PEG_RADIUS + BALL_RADIUS)
// Bars fill the space under the board; the tallest bar is always HIST_H tall
#define HIST_TOP       (BOTTOM_ROW_Y + 15)
#define HIST_LABEL_Y   (SCREEN_H - 18) // two rows of compact count labels
#define HIST_BOTTOM    (HIST_LABEL_Y - 3)
#define HIST_H         (HIST_BOTTOM - HIST_TOP + 1)
#define BAR_W         (PEG_HORZ_SEP - 4)

// [OPT 2] One row of counters per core (++ isn't atomic across cores);
// write [get_core_num()], sum both rows to read
int bins_core[2][NUM_BINS] ;
int fallen_core[2] ;

// [PROMPT #11, #13] "When the ball exits the bottom of the screen, it should
// automatically drop again from the top." Also called for every respawn.
// [PROMPT #19, #20] "does this code avoid the case where a ball might spawn
// with 0 x velocity and stay stuck bouncing on the first peg?" / "what does it
// mean to force it odd" -> OR with 1 sets the low bit, so vx can never be 0.
// Drop a ball just above the top peg with a small random vx.
// [OPT 7] vx is forced odd (never 0, which would balance on the top peg
// forever) and x gets +/-0.5 px jitter, keeping enough distinct paths at fix10.
void spawnBall(ball_t* b)
{
    fix15 x = int2fix15(SCREEN_W / 2) + (fix15)((rand() & 0x7FFF) - 0x4000) ; // 0x4000 = 0.5 px
    fix15 vx = vel2fix15(((rand() & 0x1FF) - 0x100) | 1) ; // 0x100 = 0.25 at fix10
    ballStore(b, x, int2fix15(BOARD_TOP_Y - 50), vx, 0, MAKE_META(-1, 0)) ; // no peg yet, not binned
}

// Count a ball in the bin it passed through (the caller marks it binned)
void __not_in_flash_func(binBall)(fix15 x)
{
    int xi = fix2int15(x) - BOTTOM_LEFT_X ; // relative to leftmost bottom peg
    int bin = (xi < 0) ? 0 : (xi / PEG_HORZ_SEP) + 1 ; // left of it = bin 0
    if (bin > NUM_BINS - 1) bin = NUM_BINS - 1 ; // right of rightmost peg
    int core = get_core_num() ; // [OPT 2] this core's own counters
    bins_core[core][bin]++ ;
    fallen_core[core]++ ;
}

// [PROMPT #21] "Below each histogram bar can you write the current value of
// each bar?"
// Compact 3x5 digits keep exact counts readable beneath the narrow bins.
// Alternate rows so even a ten-digit count cannot overlap its neighbors.
static void drawBinCount(int bin, int count)

```

```

{
    // One byte per pixel row of the glyph; the low 3 bits are the 3 columns
    // (bit 2 = leftmost), e.g. '0' = 111 / 101 / 101 / 101 / 111.
    static const uint8_t digits[10][5] = {
        {7, 5, 5, 5, 7}, {2, 6, 2, 2, 7}, {7, 1, 7, 4, 7},
        {7, 1, 7, 1, 7}, {5, 5, 7, 1, 1}, {7, 4, 7, 1, 7},
        {7, 4, 7, 5, 7}, {7, 1, 1, 1, 1}, {7, 5, 7, 5, 7},
        {7, 5, 7, 1, 7}
    };
    char label[12]; // fits any 32-bit int plus the terminator
    int length = snprintf(label, sizeof(label), "%d", count);
    int width = length * 4 - 1; // 3 px per digit + 1 px gap
    int x = BIN0_CENTER_X + bin * PEG_HORZ_SEP - width / 2; // centered under the bar
    x = MIN(MAX(x, 0), SCREEN_W - width); // keep the outer labels on screen
    int y = HIST_LABEL_Y + (bin & 1) * 8; // odd bins drop to the second row
    for (int i = 0; i < length; i++) {
        int digit = label[i] - '0';
        for (int row = 0; row < 5; row++) {
            for (int col = 0; col < 3; col++) {
                if (digits[digit][row] & (1u << (2 - col))) {
                    drawPixel(x + i * 4 + col, y + row, WHITE);
                }
            }
        }
    }
}

// Draw the histogram under the board, scaled so the fullest bin is HIST_H tall
void drawHistogram()
{
    int bins[NUM_BINS]; // [OPT 2] sum of both cores' counts
    int max_count = 0;
    for (int k = 0; k < NUM_BINS; k++) {
        bins[k] = bins_core[0][k] + bins_core[1][k];
        if (bins[k] > max_count) max_count = bins[k];
    }
    for (int k = 0; k < NUM_BINS; k++) drawBinCount(k, bins[k]);
    if (max_count == 0) return; // zero labels remain visible after a reset

    for (int k = 0; k < NUM_BINS; k++) {
        int h = bins[k] * HIST_H / max_count;
        if (h == 0) continue;
        // bar centered on its gap, clipped to the screen at the two outer bins
        int left = BIN0_CENTER_X + k * PEG_HORZ_SEP - BAR_W / 2;
        int right = left + BAR_W;
        if (left < 0) left = 0;
        if (right > SCREEN_W) right = SCREEN_W;
        fillRect(left, HIST_BOTTOM - h + 1, right - left, h, GREEN);
    }
}

// =====
// === DMA =====
// =====
// Peg sound: a precomputed table streamed to the DAC by two chained DMA
// channels, so the CPU does no audio work at all.
#define PIN_CS 13
#define PIN_SCK 14
#define PIN_MOSI 15

```

```

#define SPI_PORT spi1
// A-channel, 1x gain, active
#define DAC_config_chan_A 0b0011000000000000

#define SOUND_FS      50000    // DAC sample rate (Hz)
#define PEG_SOUND_FREQ 800.0f  // pitch of the "thunk" (Hz)
#define PEG_SOUND_LEN  500     // 500 samples / 50 kHz = 10 ms (short, so more hits get their own sound)
#define PEG_ATTACK     50      // samples to ramp up (avoids a click at the start)

// Precomputed sound, each sample already has the DAC config bits OR'd in
static uint16_t peg_sound[PEG_SOUND_LEN] ;
// The control channel copies this POINTER (not the table) into the data channel's
// read address, which is how each trigger rewinds playback to sample 0
static uint16_t * peg_sound_addr = &peg_sound[0] ;
static int snd_data_chan ;    // streams samples to the SPI/DAC
static int snd_ctrl_chan ;    // chained: resets snd_data_chan's read address

// Set up SPI, build the sound table, and configure the DMA
void initPegSound()
{
    // SPI at 20 MHz, 16-bit transfers; CS driven by the SPI hardware
    spi_init(SPI_PORT, 20000000) ;
    spi_set_format(SPI_PORT, 16, 0, 0, 0) ;
    gpio_set_function(PIN_CS,    GPIO_FUNC_SPI) ;
    gpio_set_function(PIN_SCK,   GPIO_FUNC_SPI) ;
    gpio_set_function(PIN_MOSI,  GPIO_FUNC_SPI) ;

    // Sine with a short linear attack and a linear decay, centered on mid-scale
    for (int i = 0; i < PEG_SOUND_LEN; i++) {
        float env = (i < PEG_ATTACK) ? (float)i / PEG_ATTACK
                                     : (float)(PEG_SOUND_LEN - i) / (PEG_SOUND_LEN - PEG_ATTACK) ;
        int sample = (int)(2047.0f * env * sinf(6.2832f * PEG_SOUND_FREQ * i / SOUND_FS)) + 2048 ;
        peg_sound[i] = DAC_config_chan_A | (sample & 0x0fff) ;
    }

    // Park the DAC at mid-scale now, so the first sound doesn't start with a pop
    uint16_t mid = DAC_config_chan_A | 2048 ;
    spi_write16_blocking(SPI_PORT, &mid, 1) ;

    snd_data_chan = dma_claim_unused_channel(true) ;
    snd_ctrl_chan = dma_claim_unused_channel(true) ;

    // Control channel: rewinds the data channel to the start of the table, then chains to it
    dma_channel_config c = dma_channel_get_default_config(snd_ctrl_chan) ;
    channel_config_set_transfer_data_size(&c, DMA_SIZE_32) ;
    channel_config_set_read_increment(&c, false) ;
    channel_config_set_write_increment(&c, false) ;
    channel_config_set_chain_to(&c, snd_data_chan) ;
    dma_channel_configure(
        snd_ctrl_chan,
        &c,
        &dma_hw->ch[snd_data_chan].read_addr,    // write: data channel's read address
        &peg_sound_addr,                          // read: POINTER to the sound's address
        1,                                         // one transfer
        false) ;                                  // don't start

    // Pacing timer = sys_clk / SOUND_FS; read at runtime, so it follows the overclock [OPT 6]
    int snd_timer = dma_claim_unused_timer(true) ;
    dma_timer_set_fraction(snd_timer, 1, (uint16_t)(clock_get_hz(clk_sys) / SOUND_FS)) ;

```

```

// Data channel: table -> SPI data register, one sample per timer tick
dma_channel_config c2 = dma_channel_get_default_config(snd_data_chan) ;
channel_config_set_transfer_data_size(&c2, DMA_SIZE_16) ;
channel_config_set_read_increment(&c2, true) ;
channel_config_set_write_increment(&c2, false) ;
channel_config_set_dreq(&c2, dma_get_timer_dreq(snd_timer)) ;
dma_channel_configure(
    snd_data_chan, &c2,
    &spi_get_hw(SPI_PORT)->dr,           // write: SPI data register
    peg_sound,                           // read: start of the sound
    PEG_SOUND_LEN,                       // transfers per trigger (reloaded on each trigger)
    false) ;                             // don't start
}

// Start the peg sound; if one is already playing, drop this one (never blocks)
void __not_in_flash_func(playPegSound)()
{
    if (dma_channel_is_busy(snd_data_chan) || dma_channel_is_busy(snd_ctrl_chan)) return ;
    dma_start_channel_mask(1u << snd_ctrl_chan) ;
}

// =====
// [OPT 4] CHEAPER COLLISION MATH (in updateBall below)
// Bottleneck : each contact test paid a sqrt, each collision two 64-bit
//               software divides (divfix), every substep re-divided v by
//               the step count, and balls below the board still ran the
//               full substep + peg search.
// Approach    : (a) compare SQUARED distance against COLLIDE_DIST2, so
//               near-misses never take a sqrt; (b) one hardware-FPU
//               1/sqrtf() per real collision, multiplied into dx and dy;
//               (c) compute step_vx/step_vy once per frame; (d) fast path:
//               below BIN_LINE_Y there are no pegs, so just x += vx.
// Effect      : (d) alone skips the peg search for about half the balls.
// Expected / measured : modest gain on top of OPT 3;
//               5,000 -> 5,200 balls at 150 MHz (+4%).
// =====
// [PROMPT #11] "...using the pseudocode above (which comes from the collision
// physics for the digital Galton Board webpage), get one ball to bounce off one
// peg... a peg is a ball with zero velocity and infinite mass." The substeps,
// the v.n guard and the pointer signature were hand-edited afterwards, and
// [PROMPT #12] fixed the resulting build errors.
// [PROMPT #16] tried the coefficient-of-restitution form dv = -(1+CR)(n.v)n;
// the final code instead reflects with -2(n.v)n and then scales the whole
// velocity by `bounciness` on a new peg (see below).
// One frame of ball physics. Peg collision: dv = -2 (n.v) n, where n is the
// unit normal from peg to ball (a peg is an infinite-mass ball at rest).
void __not_in_flash_func(updateBall)(ball_t* b)
{
    // [OPT 7] Unpack the 12-byte ball into locals; packed back at the end
    fix15 x      = BALL_POS(b->xm) ;
    fix15 y      = BALL_POS(b->ym) ;
    fix15 vx     = vel2fix15(b->vx) ;
    fix15 vy     = vel2fix15(b->vy) ;
    int meta     = BALL_META(b) ;
    int last_peg = META_LAST_PEG(meta) ;
    int binned   = META_BINNED(meta) ;

    // [OPT 4d] Below the board: no pegs left to hit, skip substeps and peg search

```

```

if (y > BIN_LINE_Y) {
    x = x + vx ;
    y = y + vy ;
} else {
    // Substeps of at most ~4 px, so a fast ball can't skip past or sink into a peg
    int speed = fix2int15(MAX(absfix15(vx), absfix15(vy))) ;
    int steps = (speed >> 2) + 1 ;
    fix15 step_vx = vx / steps ;    // [OPT 4c] divide once, not every substep
    fix15 step_vy = vy / steps ;

    for (int s = 0; s < steps; s++) {
        // Move one substep
        x = x + step_vx ;
        y = y + step_vy ;

        // [OPT 1] Only the nearest peg can be in contact
        int peg = nearestPeg(x, y) ;
        if (peg < 0) continue ;
        fix15 dx = x - peg_x[peg] ;
        fix15 dy = y - peg_y[peg] ;

        // Bounding box first, then [OPT 4a] squared distance: no sqrt for near-misses
        if ((absfix15(dx) < COLLIDE_DIST) && (absfix15(dy) < COLLIDE_DIST)) {
            fix15 dist2 = multfix15(dx,dx) + multfix15(dy,dy) ;

            // dist2 > 0 guards the divide if the ball lands exactly on the peg center
            if ((dist2 < COLLIDE_DIST2) && (dist2 > 0)) {
                // [OPT 4b] Unit normal via one FPU 1/sqrt (was two software divfix); dx*inv is already fix15
                float inv_dist = 1.0f / sqrtf(fix2float15(dist2)) ;
                fix15 normal_x = (fix15)(dx * inv_dist) ;
                fix15 normal_y = (fix15)(dy * inv_dist) ;

                // Velocity component along the normal: < 0 means moving INTO the peg
                fix15 v_dot_n = multfix15(normal_x, vx) + multfix15(normal_y, vy) ;

                // Teleport just outside contact range, so the next substep can't re-collide
                x = peg_x[peg] + multfix15(normal_x, TELEPORT_DIST) ;
                y = peg_y[peg] + multfix15(normal_y, TELEPORT_DIST) ;

                // Reflect only if approaching (reflecting a receding ball throws it back in)
                if (v_dot_n < 0) {
                    fix15 intermediate_term = multfix15(int2fix15(-2), v_dot_n) ;
                    vx = vx + multfix15(normal_x, intermediate_term) ;
                    vy = vy + multfix15(normal_y, intermediate_term) ;

                    // Damping and sound only on a new peg, not while rolling on the same one
                    if (peg != last_peg) {
                        playPegSound() ;
                        vx = multfix15(bounciness, vx) ;
                        vy = multfix15(bounciness, vy) ;
                        last_peg = peg ;
                    }
                }
            }
        }
    }
}
}

// Count it as it leaves the board (it still drifts sideways below)

```

```

if (!binned && (y > BIN_LINE_Y)) {
    binBall(x) ;
    binned = 1 ;
}

// [PROMPT #13] "the current code lets it double bounce when the ball touches the
// peg and bounces up and hits the invisible wall downward" -> only the real
// screen edge counts; there are no side walls, as on a real Galton board.
// Re-spawn any ball that falls thru the bottom of the SCREEN
if (y > int2fix15(SCREEN_H)) {
    spawnBall(b) ;
    return ;
}

// Apply gravity, then [OPT 7] pack the ball back
ballStore(b, x, y, vx, vy + gravity, MAKE_META(last_peg, binned)) ;
}

// Ball color (set from the serial thread); VGA palette index 1-15
char color = CYAN ;

// =====
// [OPT 2] SPLIT THE BALLS ACROSS BOTH CORES
// Bottleneck : core 0 did all the physics and drawing while core 1 sat idle.
// Approach   : split by index -- core 0 owns balls [0, split), core 1 owns
//              [split, num_balls) -- and each core updates AND draws its own
//              balls (updateAndDrawBall), so there's no lock and no sync
//              inside the frame. Per-core histogram counters remove the only
//              shared write. Semaphores form a fork/join each frame, and
//              core 1 also draws the pegs. Core0/Core1/Spare are shown on
//              screen to find the limit.
// Effect      : roughly doubles the compute available per frame.
// Expected / measured : ~2x; not measured alone (first recorded limit was
//              5,000 balls after OPT 3).
// =====
semaphore_t draw_semaphore ; // core 0 -> core 1: "new frame, start clearing your half"

// =====
// [OPT 5] PARALLEL CLEAR + AUTOMATIC SPLIT BALANCE
// Bottleneck : core 0 cleared the whole 153.6 KB buffer and drew the
//              histogram while core 1 idled (and that time was hidden from
//              core1_us); a fixed 60/40 split is wrong as balls move
//              between the peg area (substeps) and the free-fall area.
// Approach    : core 0 clears rows 0-239 while core 1 clears rows 240-479 and
//              draws the histogram (entirely in the bottom half), then a
//              two-way barrier (top/bot semaphores) so nobody draws onto an
//              uncleared half. split_pm (per-mille) moves 0.1% of the balls
//              per frame toward whichever core finished first last frame,
//              with a 50 us deadband against noise.
// Effect      : clear time halved; both cores finish together, and Spare is
//              now an honest margin.
// Expected / measured : a few hundred us per frame plus better balance;
//              shipped together with OPT 6, not measured alone.
// =====
semaphore_t top_semaphore ; // core 0 -> core 1: "top half clear, num_balls/split final"
semaphore_t bot_semaphore ; // core 1 -> core 0: "bottom half clear, histogram drawn"
semaphore_t done_semaphore ; // [OPT 2] core 1 -> core 0: "my share is drawn" (don't clear under me)

#define CLEAR_SPLIT_Y 240 // core 0 clears rows [0, 240), core 1 clears [240, 480)

```

```

#define SPLIT_PM_INIT 600    // starting guess: core 0 takes 60% of the balls
#define SPLIT_DEADBAND 50    // us: don't chase differences smaller than this
int split_pm = SPLIT_PM_INIT ; // only touched by core 0
volatile int split = 0 ;

// [OPT 2] per-core work time last frame (us); both start at vsync, so they compare finish times
volatile uint32_t core0_us = 0 ;
volatile uint32_t core1_us = 0 ;
#define FRAME_BUDGET_US 16667 // DOUBLE_BUFFER_60 -> 60 fps

// === MISSED-DEADLINE LED ===
// The VGA DMA sets start_flag at every vsync (buffer swap). If it is set again
// before both cores finish, the frame was swapped in half-drawn: deadline missed.
extern int start_flag ; // defined in vga16_graphics_v3.c
#define LED_PIN PICO_DEFAULT_LED_PIN // GPIO 25 on the Pico 2
#define LED_HOLD_FRAMES 30 // keep the LED lit ~0.5 s after a miss, so even a single missed frame is visible
static int led_hold = 0 ; // frames left to keep the LED on (core 0 only)
static int missed_frames = 0 ; // missed deadlines since boot (core 0 only)

// =====
// [OPT 3] PRECOMPUTED BALL SPRITE
// Bottleneck : the library circle call redid the same shape for every ball
//              every frame (fillCircle: 5 software sqrts + 10 drawHLine
//              calls, each with range checks and a tiny memset).
// Approach   : the radius never changes, so compute each row's half-width
//              once (initBallSprite) and write whole bytes (2 px each)
//              straight into the frame buffer. Balls touching the screen
//              edge fall back to the library call, which clips. Also: float
//              literals instead of double in the fix15 macros (M33 FPU is
//              single-precision only).
// Effect      : the biggest drawing speedup.
// Expected / measured : 5,000 balls at 150 MHz (with OPT 1 and 2).
// Balls are now drawn hollow: the sprite is a ring (disk r minus disk r-1),
// drawn as two spans per row.
// =====
extern char * current_draw_buffer ; // defined in vga16_graphics_v3.c
int32_t sqrt_i32(int32_t v) ; // defined in vga16_graphics_v3.c
static int ball_dx[BALL_RADIUS + 1] ; // outer half-width of row i above/below center
static int ball_hx[BALL_RADIUS + 1] ; // hole half-width of row i (0 = solid row)

// Ring half-widths (same formula as fillCircle); every row at least 1 px thick
void initBallSprite(void)
{
    int r2 = BALL_RADIUS * BALL_RADIUS + BALL_RADIUS ;
    int h = BALL_RADIUS - 1 ;
    int h2 = h * h + h ;
    for (int i = 0; i <= BALL_RADIUS; i++) {
        ball_dx[i] = sqrt_i32(r2 - i * i) ;
        ball_hx[i] = (i <= h) ? sqrt_i32(h2 - i * i) : 0 ;
        // keep every row's outline at least 1 px thick
        if (ball_hx[i] >= ball_dx[i]) ball_hx[i] = ball_dx[i] - 1 ;
    }
}

// Fill pixels [x, x+w) of one 640-px row (2 px/byte: even x = low nibble)
static inline void drawSpan(unsigned char* row, int x, int w, unsigned char c)
{
    if (x & 1) { // lone pixel at the left

```

```

    row[x >> 1] = (row[x >> 1] & 0x0F) | (c << 4) ;
    x++ ; w-- ;
}
unsigned char* p = row + (x >> 1) ;
unsigned char both = c | (c << 4) ;
// [OPT 8] Whole bytes written out (a span is at most 4 bytes): GCC turned a
// byte loop into memset() calls -- slow for 1-4 bytes, and memset is in flash
int n = w >> 1 ;
if (n > 0) p[0] = both ;
if (n > 1) p[1] = both ;
if (n > 2) p[2] = both ;
if (n > 3) p[3] = both ;
p += n ;
if (w & 1) *p = (*p & 0xF0) | c ;           // lone pixel at the right
}

// One row of the ring: the two sides, or the whole span where there's no hole
static inline void drawRingRow(unsigned char* row, int x0, int dx, int hx, unsigned char c)
{
    if (hx <= 0) {
        drawSpan(row, x0 - dx, 2 * dx, c) ;
    } else {
        drawSpan(row, x0 - dx, dx - hx, c) ;    // left side
        drawSpan(row, x0 + hx, dx - hx, c) ;    // right side
    }
}

// [OPT 3] Draw one hollow ball from the precomputed sprite
static inline void drawBall(int x0, int y0, char c)
{
    // Fast path only when the whole ball is on screen; drawCircle() clips the rest
    if ((x0 - BALL_RADIUS < 0) || (x0 + BALL_RADIUS > 639) ||
        (y0 - BALL_RADIUS < 0) || (y0 + BALL_RADIUS > 479)) {
        drawCircle(x0, y0, BALL_RADIUS, c) ;
        return ;
    }
    unsigned char* center = (unsigned char*)current_draw_buffer + 320 * y0 ;
    drawRingRow(center, x0, ball_dx[0], ball_hx[0], c) ;
    for (int i = 1; i <= BALL_RADIUS; i++) {
        drawRingRow(center + 320 * i, x0, ball_dx[i], ball_hx[i], c) ;    // row below center
        drawRingRow(center - 320 * i, x0, ball_dx[i], ball_hx[i], c) ;    // row above center
    }
}

// [OPT 2] Physics + draw for one ball, back to back, so the cores never sync mid-frame
// [PROMPT #22] "can you stop drawing the balls when it reaches the peg board
// bottom so they dont cover the histogram"
static inline void updateAndDrawBall(ball_t* b)
{
    updateBall(b) ;
    // Hide balls once they leave the peg board so they cannot cover the histogram.
    // Physics continues below the board until the normal bottom-of-screen respawn.
    if (BALL_POS(b->ym) <= BIN_LINE_Y) {
        drawBall(fix2int15(BALL_POS(b->xm)), fix2int15(BALL_POS(b->ym)), color) ;
    }
}

// =====
// [OPT 8] HOT CODE IN SRAM

```

```

// Bottleneck : at 13,000 balls Spare was 0. All code ran from flash
//               through ONE 16 KB XIP cache shared by both cores, so the
//               cores stalled on each other's misses; the disassembly also
//               showed 10 memset() calls per ball jumping back to flash.
// Approach    : run the per-ball path from SRAM. updateAndDrawRange() is
//               __no_inline_not_in_flash_func (no_inline is required: with
//               plain __not_in_flash_func, -Ofast inlined it back into the
//               flash-resident protothreads) and both cores call it, so the
//               inline draw helpers land in SRAM with it. updateBall,
//               nearestPeg, binBall and playPegSound are __not_in_flash_func.
//               drawSpan's loop was replaced by direct stores.
// Effect      : the per-ball path makes no flash calls except drawCircle
//               (edge balls only) and rand() (respawns). Costs ~2 KB of RAM.
// Expected / measured : expected +5-20%; measured 13,000 balls at 0 us
//               spare -> 16,000 balls with ~600 us spare (about +25%).
// =====
void __no_inline_not_in_flash_func(updateAndDrawRange)(int from, int to)
{
    for (int i = from; i < to; i++) updateAndDrawBall(&balls[i]) ;
}

// =====
// === users serial input thread
// =====
// Reads a color (1-15) from the serial port and applies it to the balls
// (from the original demo; runs on core 0 alongside protothread_anim)
static PT_THREAD (protothread_serial(struct pt *pt))
{
    // Mark beginning of thread
    PT_BEGIN(pt);
    // stores user input
    static int user_input ;
    // wait 1 s before printing the banner
    PT_YIELD_usec(1000000) ;
    // announce the threader version
    sprintf(pt_serial_out_buffer, "Protothreads RP2040 v1.4\n\r");
    // non-blocking write
    serial_write ;
    while(1) {
        // print prompt
        sprintf(pt_serial_out_buffer, "input a number in the range 1-15: ");
        // non-blocking write
        serial_write ;
        // spawn a thread to do the non-blocking serial read
        serial_read ;
        // convert input string to number
        sscanf(pt_serial_in_buffer, "%d", &user_input) ;
        // update ball color (ignore anything outside the 15 valid palette entries)
        if ((user_input > 0) && (user_input < 16)) {
            color = (char)user_input ;
        }
    } // END WHILE(1)
    PT_END(pt);
} // serial thread

// Animation on core 0: frame control, knob handling, its share of the balls, text
static PT_THREAD (protothread_anim(struct pt *pt))
{
    // Mark beginning of thread

```

```

PT_BEGIN(pt);

// Build the peg table and the ball sprite (balls are added in the loop)
initPegs() ;
initBallSprite() ;

static char text_str[40] ; // static: pthread locals must survive PT_YIELD / PT_WAIT

while(1) {
    // Wait for vsync (the DMA has just swapped buffers)
    PT_YIELD_UNTIL(pt, draw_start_signal()) ;
    // [OPT 2] time core 0's share of the frame
    static uint32_t t0 ;
    t0 = time_us_32() ;

    // A knob click changed a parameter: reset the histogram and fallen count.
    // Safe here: core 1 hasn't been woken yet, so nothing is reading or counting.
    if (stats_reset) {
        stats_reset = 0 ;
        memset(bins_core, 0, sizeof(bins_core)) ;
        memset(fallen_core, 0, sizeof(fallen_core)) ;
    }

    // [OPT 5] Wake core 1 now, so both cores clear their half in parallel
    PT_SEM_SDK_SIGNAL(pt, &draw_semaphore) ;
    clearRegion(0, CLEAR_SPLIT_Y, BLACK) ;

    // Knob bookkeeping, before core 1 is released to draw (it then sees the
    // final num_balls/split and never touches a ball being re-spawned)
    pollButton() ; // push button: next knob mode on press
    int target = rot_counter ; // read the ISR's value once
    // Adding balls: all new ones spawn at the same spot above the top peg
    for (int i = num_balls; i < target; i++) {
        spawnBall(&balls[i]) ;
    }
    // Removing balls: the extras just stop being updated/drawn
    num_balls = target ;
    // [OPT 5] Auto-balance: shift 0.1% of the balls toward the core that finished first
    int diff = (int)core1_us - (int)core0_us ;
    if (diff > SPLIT_DEADBAND) split_pm = MIN(split_pm + 1, 1000) ;
    else if (diff < -SPLIT_DEADBAND) split_pm = MAX(split_pm - 1, 0) ;
    // [OPT 2] decide which balls each core owns this frame
    split = (num_balls * split_pm) / 1000 ;

    // [OPT 5] Barrier: release core 1, then wait for the bottom half and the histogram
    PT_SEM_SDK_SIGNAL(pt, &top_semaphore) ;
    PT_SEM_SDK_WAIT(pt, &bot_semaphore) ;

    // [OPT 2][OPT 8] Core 0's share of the balls: [0, split), run from SRAM
    updateAndDrawRange(0, split) ;

    // === TEXT === (core times shown are last frame's; core0_us includes the text)
    // [PROMPT #5, #6, #9] the original "Count: %d" encoder readout (sprintf
    // kept after #8's sprintf-free version was reverted) grew into this panel.
    // Each field is drawn with a black background so it overwrites last frame's text.
    sprintf(text_str, "Balls: %d", num_balls) ;
    drawTextVGA437(10, 10, text_str, WHITE, BLACK) ;
    sprintf(text_str, "Fallen: %d", fallen_core[0] + fallen_core[1]) ;
    drawTextVGA437(10, 30, text_str, WHITE, BLACK) ;

```

```

sprintf(text_str, "Time:  %d s", (int)(time_us_64() / 1000000)) ;
drawTextVGA437(10, 50, text_str, WHITE, BLACK) ;
// [OPT 2] Per-core times and Spare (set by the slower core; ~0 = at the ball limit)
static uint32_t slowest ;
slowest = MAX(core0_us, core1_us) ;
sprintf(text_str, "Core0: %5d us", (int)core0_us) ;
drawTextVGA437(10, 70, text_str, WHITE, BLACK) ;
sprintf(text_str, "Core1: %5d us", (int)core1_us) ;
drawTextVGA437(10, 90, text_str, WHITE, BLACK) ;
sprintf(text_str, "Spare: %5d us", (int)FRAME_BUDGET_US - (int)slowest) ;
drawTextVGA437(10, 110, text_str, WHITE, BLACK) ;
// Knob mode and parameters (mode names padded so a shorter one overwrites a longer one)
static const char* const mode_name[ROT_NUM_MODES] = { "BALLS  ", "BOUNCE ", "GRAVITY" } ;
sprintf(text_str, "Knob:  %s", mode_name[rot_mode]) ;
drawTextVGA437(10, 130, text_str, WHITE, BLACK) ;
sprintf(text_str, "Bounce: %d.%02d", bounce_pct / 100, bounce_pct % 100) ;
drawTextVGA437(10, 150, text_str, WHITE, BLACK) ;
sprintf(text_str, "Gravity: %d.%02d", grav_pct / 100, grav_pct % 100) ;
drawTextVGA437(10, 170, text_str, WHITE, BLACK) ;
// [OPT 5] core 0's current share of the balls
// sprintf(text_str, "Split: %3d.%d%%", split_pm / 10, split_pm % 10) ;
// drawTextVGA437(10, 190, text_str, WHITE, BLACK) ;
// sprintf(text_str, "Missed: %d", missed_frames) ;
// drawTextVGA437(10, 210, text_str, WHITE, BLACK) ;
core0_us = time_us_32() - t0 ;

// [OPT 2] Join: wait for core 1, so the next clear can't wipe a buffer it's drawing
PT_SEM_SDK_WAIT(pt, &done_semaphore) ;

// Missed-deadline check: vsync already flagged again = this frame was late.
// Only peek; draw_start_signal() consumes the flag to start the next frame.
if (*(volatile int*)&start_flag) {
    missed_frames++ ;
    led_hold = LED_HOLD_FRAMES ;
} else if (led_hold > 0) {
    led_hold-- ;
}
gpio_put(LED_PIN, led_hold > 0) ;

// NEVER exit while
} // END WHILE(1)
PT_END(pt);
} // animation thread

```

// Animation on core 1: bottom-half clear, histogram, pegs, its share of the balls

```
static PT_THREAD (protothread_anim1(struct pt *pt))
```

```
{
```

```
    // Mark beginning of thread
```

```
    PT_BEGIN(pt);
```

```
    // Runs on core 1 only. Per-frame order: clear bottom half -> histogram ->
```

```
    // barrier -> pegs -> its share of the balls -> signal done.
```

```
    while(1) {
```

```
        // [OPT 2] Fork: wait for core 0 to start the frame
```

```
        PT_SEM_SDK_WAIT(pt, &draw_semaphore) ;
```

```
        static uint32_t t1 ;
```

```
        t1 = time_us_32() ;
```

```

// [OPT 5] Clear the bottom half while core 0 clears the top, then draw the
// histogram (bins are stable: core 0 isn't counting balls yet)
clearLowFrame(CLEAR_SPLIT_Y, BLACK) ;
drawHistogram() ;

// [OPT 5] Barrier: bottom done; wait for the top half and final num_balls/split
PT_SEM_SDK_SIGNAL(pt, &bot_semaphore) ;
PT_SEM_SDK_WAIT(pt, &top_semaphore) ;

// [OPT 2] Hollow pegs, drawn by core 1. Balls never overlap pegs
// (TELEPORT_DIST), so drawing order between the cores doesn't matter.
for (int i = 0; i < NUM_PEGS; i++) {
    drawCircle(fix2int15(peg_x[i]), fix2int15(peg_y[i]), PEG_RADIUS, WHITE) ;    // hollow
}

// [OPT 2][OPT 8] Core 1's share of the balls: [split, num_balls), run from SRAM
updateAndDrawRange(split, num_balls) ;

core1_us = time_us_32() - t1 ;
// [OPT 2] Join: tell core 0 this share is done
PT_SEM_SDK_SIGNAL(pt, &done_semaphore) ;
} // END WHILE(1)
PT_END(pt);
} // animation thread

// =====
// === core 1 main -- started in main below
// =====
void core1_main(){
    // Add animation thread
    pt_add_thread(protothread_anim1);
    // Start the scheduler
    pt_schedule_start ;
}

// =====
// === main
// =====
// USE ONLY C-sdk library
int main(){
    // =====
    // [OPT 6] OVERCLOCK 150 -> 300 -> 350 MHz
    // Bottleneck : CPU time. At 150 MHz Spare hit 0 at 5,200 balls.
    // Approach   : raise the core voltage first (the RP2350 does not do it
    //               automatically; 1.30 V is the max without unlocking the
    //               regulator), then the system clock. VGA needs a fixed
    //               25 MHz pixel clock, so with sys_clk = D x 25 MHz:
    //               hsync/vsync clkdiv = D, rgb.pio holds = D-1 / D-3
    //               (D = 14 at 350 MHz; see the VGA/*.pio files). The sound
    //               DMA timer reads clock_get_hz(), so it adapts by itself.
    // 350 MHz PLL : needs a 1050 MHz VCO (/3 /1), which isn't a multiple of
    //               the 12 MHz crystal, so set_sys_clock_khz() (reference
    //               divider 1 only) can't produce it and would halt at boot.
    //               Program the PLL directly with reference divider 2:
    //               6 MHz x 175 = 1050 MHz -- same steps as set_sys_clock_pll().
    // Expected / measured : 300 MHz: ~2x CPU; measured 3,150 us spare at
    //               10,000 balls (RAM, not CPU, was the limit -> OPT 7).
    //               350 MHz: +17%; measured 13,000 balls at 0 us spare

```

```
// (with the 12-byte ball and hollow drawing).
// =====
vreg_set_voltage(VREG_VOLTAGE_1_30) ;
busy_wait_us(10000) ; // let the voltage settle before raising the clock
// Run clk_sys from the 48 MHz USB PLL while the sys PLL is reprogrammed
clock_configure_undivided(clk_sys,
    CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
    CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB,
    USB_CLK_HZ) ;
pll_init(pll_sys, 2, 1050 * MHz, 3, 1) ;
clock_configure_undivided(clk_sys,
    CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
    CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS,
    350 * MHz) ;
// Peripherals (UART, SPI) on the fixed 48 MHz USB PLL, as set_sys_clock_khz() did
clock_configure_undivided(clk_peri, 0,
    CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB,
    USB_CLK_HZ) ;
// initialize stdio (USB/UART serial for the color prompt)
stdio_init_all() ;

// initialize VGA (PIO state machines, DMA chain, frame buffers)
initVGA() ;

// initialize the DAC + DMA for the peg sound
initPegSound() ;

// [PROMPT #1-#10] rotary encoder setup, final form from #10:
// interrupt on BOTH pins so rot_ISR sees every step of the 4-state sequence
// === ROTARY ENCODER === A/B interrupt on both edges, with pull-ups
gpio_init(ROT_A);
gpio_init(ROT_B);
gpio_set_dir(ROT_A, GPIO_IN);
gpio_set_dir(ROT_B, GPIO_IN);
gpio_pull_up(ROT_A);
gpio_pull_up(ROT_B);
rot_prev_state = (gpio_get(ROT_A) << 1) | gpio_get(ROT_B) ;
gpio_set_irq_enabled_with_callback(ROT_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true, &rot_ISR);
gpio_set_irq_enabled(ROT_B, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true);
// Push button: active high, so pull down. Polled by pollButton(), no IRQ.
gpio_init(ROT_SW);
gpio_set_dir(ROT_SW, GPIO_IN);
gpio_pull_down(ROT_SW);
gpio_set_input_enabled(ROT_SW, false); // E9 workaround: see pollButton()

// Missed-deadline LED (on-board), off until a frame misses
gpio_init(LED_PIN);
gpio_set_dir(LED_PIN, GPIO_OUT);
gpio_put(LED_PIN, 0);

// Semaphores (initial count 0, max 1): [OPT 2] fork/join, [OPT 5] clear barrier
sem_init(&draw_semaphore, 0, 1) ;
sem_init(&top_semaphore, 0, 1) ;
sem_init(&bot_semaphore, 0, 1) ;
sem_init(&done_semaphore, 0, 1) ;

// start core 1 (semaphores must exist first, core 1 waits on draw_semaphore)
multicore_reset_core1();
multicore_launch_core1(&core1_main);
```

```
// add threads (core 0: serial input + animation; core 1: see core1_main)
pt_add_thread(protothread_serial);
pt_add_thread(protothread_anim);

// start scheduler (never returns)
pt_schedule_start ;
}
```

Appendix B: AI Prompt Log

Claude Code Session Log — Rotary Encoder + Galton Board

A record of the prompts given to Claude Code in this session, a summary of what was produced in response, and the code that came with each response, for `animation.c` in this lab.

Note: several code blocks below were later edited, replaced, or reverted by later prompts in this same session (e.g. the `sprintf`-free version in #8 was reverted in #9; the simple single-edge ISR in #5 was replaced by the full state-machine ISR in #10). Those blocks are reproduced here as they were *at that point in the conversation*, not necessarily as they appear in `animation.c` today. Blocks marked "(current)" were pulled directly from the file as it stands right now.

1. Initial rotary encoder request

Prompt:

Read the rotary encoder page and write a software interface to the rotary encoder. You should display a number on the VGA display that increments when you rotate the encoder clockwise, and decrements when you rotate it counterclockwise. do not acutally impekenmt the code in the fiel yet. see where we hae started with edit fo draw Textvga 347

Response: Read vanhunteradams.com's Rotary Encoder page. Explained the encoder hardware (pins A/B/COM, pull-ups needed, states float high/short low), the 4-state quadrature sequence for clockwise vs. counterclockwise rotation, and designed a lookup-table-based ISR that only counts a full detent click (cancels out contact bounce). Pointed out that the existing `drawTextVGA437(50, 50, "hi", char color, char bakgnd_color)` call at the time didn't compile. No file edits made yet, per instructions — code given as reference only.

```
#define ENC_A 12
#define ENC_B 11

volatile int encoder_count = 0;    // written in ISR, read by anim thread
static uint8_t enc_prev = 3;      // start at detent (both high)
static int8_t enc_accum = 0;

// index = (prev<<2)|curr ; +1 = CW step, -1 = CCW step, 0 = none/invalid
static const int8_t enc_table[16] = {
```

```

/* prev=0 */ 0, -1, +1, 0,
/* prev=1 */ +1, 0, 0, -1,
/* prev=2 */ -1, 0, 0, +1,
/* prev=3 */ 0, +1, -1, 0
};

void encoder_isr(uint gpio, uint32_t events) {
    uint8_t curr = (gpio_get(ENC_A) << 1) | gpio_get(ENC_B);
    enc_accum += enc_table[(enc_prev << 2) | curr];
    enc_prev = curr;
    if (curr == 3) { // back at detent
        if (enc_accum >= 4) encoder_count++;
        else if (enc_accum <= -4) encoder_count--;
        enc_accum = 0;
    }
}

// in main(), before multicore_launch_core1:
gpio_init(ENC_A); gpio_set_dir(ENC_A, GPIO_IN); gpio_pull_up(ENC_A);
gpio_init(ENC_B); gpio_set_dir(ENC_B, GPIO_IN); gpio_pull_up(ENC_B);
gpio_set_irq_enabled_with_callback(ENC_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &encoder_isr);
gpio_set_irq_enabled(ENC_B, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true);

static char enc_str[32];
sprintf(enc_str, "Count: %d", encoder_count);
drawTextVGA437(50, 50, enc_str, WHITE, BLACK);

```

2. Clarifying the state encoding

Prompt:

are you sure its static 8 int where is the 3 coming from?

Response: Explained `uint8_t` vs. `int`, why `static` is needed (the ISR needs the previous state to persist between calls), and walked through why the rest state is 3 — both A and B read high (1,1) when neither is touching a contact pad, assuming pull-ups are enabled, giving $(1 \ll 1) | 1 = 3$. Flagged that this assumes the encoder boots at a detent, and suggested reading the pins once at startup instead of hard-coding 3:

```
enc_prev = (gpio_get(ENC_A) << 1) | gpio_get(ENC_B);
```

No other code given this turn.

3. First code drop

Prompt:

ok provide me jsut the code to add in

Response: Provided the full code to paste in by hand (not yet applied to the file directly — given as text for the user to add themselves).

Globals / ISR, to go with the other globals:

```
// === Rotary encoder ===
#define ENC_A 12
#define ENC_B 11

volatile int encoder_count = 0; // written in ISR, read by anim thread
static uint8_t enc_prev = 3; // previous (A<<1)|B state
static int8_t enc_accum = 0; // quarter-steps since last detent

// index = (prev<<2)|curr ; +1 = CW step, -1 = CCW step, 0 = none/invalid
static const int8_t enc_table[16] = {
/* prev=0 */ 0, -1, +1, 0,
/* prev=1 */ +1, 0, 0, -1,
/* prev=2 */ -1, 0, 0, +1,
/* prev=3 */ 0, +1, -1, 0
};

// GPIO interrupt: runs on any edge of A or B
void encoder_isr(uint gpio, uint32_t events) {
    uint8_t curr = (gpio_get(ENC_A) << 1) | gpio_get(ENC_B);
    enc_accum += enc_table[(enc_prev << 2) | curr];
    enc_prev = curr;
    if (curr == 3) { // back at detent
        if (enc_accum >= 4) encoder_count++;
        else if (enc_accum <= -4) encoder_count--;
        enc_accum = 0;
    }
}
```

Replacing the broken `drawTextVGA437` line in `protothread_anim`:

```
// draw rotary text
static char enc_str[32];
sprintf(enc_str, "Count: %d", encoder_count);
drawTextVGA437(50, 50, enc_str, WHITE, BLACK);
```

In `main()`, after `initVGA()` and before `multicore_launch_core1`:

```
// initialize rotary encoder
gpio_init(ENC_A); gpio_set_dir(ENC_A, GPIO_IN); gpio_pull_up(ENC_A);
gpio_init(ENC_B); gpio_set_dir(ENC_B, GPIO_IN); gpio_pull_up(ENC_B);
enc_prev = (gpio_get(ENC_A) << 1) | gpio_get(ENC_B);
gpio_set_irq_enabled_with_callback(ENC_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &encoder_isr);
```

```
gpio_set_irq_enabled(ENC_B, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true);
```

4. Simplify / explain like I'm 5

Prompt (after a model switch to Sonnet 5):

is there some way to make it simpler? wht does 0 through 3 mean? explain to me like 5 year old

Response: Explained the 0–3 state numbers as just "nicknames" for the four combinations of two on/off switches (A and B). Offered a simpler single-edge-triggered ISR (interrupt on A only, compare to B to get direction) as an easier starting point, with the trade-off that it's more sensitive to bounce than the 4-state table version.

```
#define ENC_A 12
#define ENC_B 11

volatile int encoder_count = 0;

// Runs only when A changes.
void encoder_isr(uint gpio, uint32_t events) {
    if (gpio_get(ENC_A) == gpio_get(ENC_B)) {
        encoder_count++; // clockwise
    } else {
        encoder_count--; // counterclockwise
    }
}

gpio_init(ENC_A); gpio_set_dir(ENC_A, GPIO_IN); gpio_pull_up(ENC_A);
gpio_init(ENC_B); gpio_set_dir(ENC_B, GPIO_IN); gpio_pull_up(ENC_B);
gpio_set_irq_enabled_with_callback(ENC_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &encoder_isr);
```

5. "Drop in the code"

Prompt:

drop in the code

Response: First real edits to `animation.c` (the user had already hand-added a broken `void rot_ISR()` with no body and incomplete GPIO setup; these edits fixed and completed it):

Fixed ISR + globals (replacing the broken block):

```
// rotary definitions
```

```

#define ROT_A 12
#define ROT_B 11
volatile int rot_counter = 0;

// uS per frame
#define FRAME_RATE 33000

// Fires on every edge of ROT_A. Peek at ROT_B to tell direction.
void rot_ISR(uint gpio, uint32_t events)
{
    if (gpio_get(ROT_A) == gpio_get(ROT_B)) {
        rot_counter++ ; // clockwise
    } else {
        rot_counter-- ; // counterclockwise
    }
}

```

Added to the GPIO init block in `main()` (pull-ups and interrupt hookup had been missing):

```

gpio_pull_up(ROT_A);
gpio_pull_up(ROT_B);
gpio_set_irq_enabled_with_callback(ROT_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &rot_ISR);

```

Replaced the static "hi" text in `protothread_anim` with a live counter:

```

static char rot_str[32] ;
sprintf(rot_str, "Count: %d", rot_counter) ;
drawTextVGA437(50, 50, rot_str, WHITE, BLACK);

```

6. Comment the prompts and edited sections

Prompt:

please make sure to comment the prompt used and also comment the specific sections that you've edited or implemented

Response: Added block comments quoting the originating prompt above the rotary encoder code, and wrapped every edited region in `=== ... === / === END ... ===` markers. (This is the state of the file right after this turn's edits, before the user's own further edits and before the v2 quadrature rewrite in #10.)

```

// =====
// === ROTARY ENCODER INTERFACE (added w/ Claude Code assistance)
// Prompt: "Read the rotary encoder page and write a software interface
// to the rotary encoder. You should display a number on the VGA display
// that increments when you rotate the encoder clockwise, and decrements

```

```
// when you rotate it counterclockwise."
//
// Approach: GPIO A/B float high via pull-ups and short low when a
// contact pad touches them (see vanhunteradams.com Rotary_Encoder page).
// We interrupt on every edge of A only, and read B at that instant to
// tell which direction the disk is turning (classic 2-pin quadrature
// direction check). rot_counter is the displayed value.
// =====
#define ROT_A 12
#define ROT_B 11
volatile int rot_counter = 0;

// Fires on every edge of ROT_A. Peek at ROT_B to tell direction.
void rot_ISR(uint gpio, uint32_t events)
{
    if (gpio_get(ROT_A) == gpio_get(ROT_B)) {
        rot_counter++ ;    // clockwise
    } else {
        rot_counter-- ;    // counterclockwise
    }
}
// === END ROTARY ENCODER ISR =====

// === ROTARY ENCODER: GPIO + interrupt setup (Claude Code assisted) ===
// Pull-ups needed because A/B float when not touching a contact pad.
// Interrupt registered on A's edges only; rot_ISR reads B for direction.
gpio_init(ROT_A);
gpio_init(ROT_B);
gpio_set_dir(ROT_A, GPIO_IN);
gpio_set_dir(ROT_B, GPIO_IN);
gpio_pull_up(ROT_A);
gpio_pull_up(ROT_B);
gpio_set_irq_enabled_with_callback(ROT_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &rot_ISR);
// === END ROTARY ENCODER SETUP =====

// === ROTARY ENCODER: display rot_counter (Claude Code assisted) ===
static char rot_str[32] ;
sprintf(rot_str, "Count: %d", rot_counter) ;
drawTextVGA437(50, 50, rot_str, WHITE, BLACK);
// === END ROTARY ENCODER DISPLAY =====
```

7. Diagnosing inconsistent increments (discussion only)

Prompt:

okay now i notice that we incrementing or decrementing by inconsistent large amounts, i want it to be less sensitive increase or decrease only one at a time...? talk me through how i would do this first

Response: Explained the root cause without editing code yet: the single-edge approach reacts to every edge of A, and one physical click already produces 2 edges on A, plus contact bounce adds more. Walked through the fix — tracking the full 4-state sequence with an accumulator that only updates the display when a complete click (± 4 steps) returns to the rest state — and why that's naturally bounce-immune. No code given this turn; the fix was implemented in #10.

8. Printing without `sprintf`

Prompt:

can you do the printign to vga without sprit f plez

Response: Added a manual `intToStr()` helper (digit-by-digit conversion, no `stdio` formatting) and switched the display code to use it. (This was reverted in #9, so it no longer appears in the file.)

```
// Convert an int to a decimal string in buf, without sprintf.
// Handles negative numbers. buf must be large enough (12 bytes is safe
// for any 32-bit int, sign + digits + null terminator).
void intToStr(int n, char* buf)
{
    int i = 0 ;
    int negative = (n < 0) ;
    unsigned int mag = negative ? (unsigned int)(-n) : (unsigned int)n ;

    // peel off digits least-significant-first
    do {
        buf[i++] = (char)('0' + (mag % 10)) ;
        mag /= 10 ;
    } while (mag > 0) ;

    if (negative) buf[i++] = '-' ;
    buf[i] = '\0' ;

    // reverse in place
    for (int lo = 0, hi = i - 1 ; lo < hi ; lo++, hi--) {
        char tmp = buf[lo] ;
        buf[lo] = buf[hi] ;
        buf[hi] = tmp ;
    }
}

// === ROTARY ENCODER: display rot_counter (Claude Code assisted) ===
// No sprintf -- convert manually and draw label + number separately.
intToStr(rot_counter, rot_str) ;
drawTextVGA437(50, 50, "Count: ", WHITE, BLACK);
drawTextVGA437(120, 50, rot_str, WHITE, BLACK);
```

9. Revert back to `sprintf`

Prompt:

nvm go back to sprintf

Response: Reverted the display line to `sprintf` and removed the now-unused `intToStr()` helper entirely.

```
// === ROTARY ENCODER: display rot_counter (Claude Code assisted) ===
sprintf(rot_str, "Count: %d", rot_counter) ;
drawTextVGA437(50, 50, rot_str, WHITE, BLACK);
```

10. Implementing the full quadrature state machine

Prompt:

okay great, onething i am noticing, i feel the brush a provide one incrmnt and brush b provide another incrmnt. it is as is one clockwise rotiaton full incrmnets by 2. maybe you do have to read a then b and keep track of the sequenc eof events

Response: Replaced the single-edge ISR with a full 4-state quadrature detector: a `rot_table[16]` lookup keyed on `(prev_state<<2)|curr_state`, an accumulator (`rot_accum`) that only commits to `rot_counter` when the state returns to rest having taken a full ± 4 -step trip, and interrupts registered on both A and B (not just A). Also seeded `rot_prev_state` from a real pin read at boot instead of assuming rest. **(current — this is still in the file today, aside from the user's own pin-number swap.)**

```
#define ROT_A 12
#define ROT_B 11
volatile int rot_counter = 0;
static volatile uint8_t rot_prev_state = 3 ; // (A<<1)|B ; 3 = rest (both high)
static volatile int8_t rot_accum = 0 ; // steps taken since last rest

// Lookup table: index = (prev_state<<2)|curr_state.
// +1 = one valid step clockwise, -1 = one valid step counterclockwise,
// 0 = no change, or a state jump that skips a step (bounce/noise) -- ignore.
static const int8_t rot_table[16] = {
/* prev=0(00) */ 0, -1, +1, 0,
/* prev=1(01) */ +1, 0, 0, -1,
/* prev=2(10) */ -1, 0, 0, +1,
/* prev=3(11) */ 0, +1, -1, 0
} ;

// Fires on every edge of EITHER A or B.
void rot_ISR(uint gpio, uint32_t events)
{
```

```

uint8_t curr_state = (gpio_get(ROT_A) << 1) | gpio_get(ROT_B) ;
rot_accum += rot_table[(rot_prev_state << 2) | curr_state] ;
rot_prev_state = curr_state ;

if (curr_state == 3) {          // landed back at rest (a full click, or none)
    if (rot_accum >= 4)          rot_counter++ ; // completed one CW click
    else if (rot_accum <= -4)    rot_counter-- ; // completed one CCW click
    rot_accum = 0 ;              // discard partial turns / bounce
}
}

// === ROTARY ENCODER: GPIO + interrupt setup (Claude Code assisted) ===
// Pull-ups needed because A/B float when not touching a contact pad.
// v2: interrupt on BOTH pins now, since rot_ISR needs to see every step
// of the 4-state sequence, not just A's edges.
gpio_init(ROT_A);
gpio_init(ROT_B);
gpio_set_dir(ROT_A, GPIO_IN);
gpio_set_dir(ROT_B, GPIO_IN);
gpio_pull_up(ROT_A);
gpio_pull_up(ROT_B);
// seed rot_prev_state with the real, current pin reading (assumes we boot
// at rest -- fine since the encoder detents there and holds until turned)
rot_prev_state = (gpio_get(ROT_A) << 1) | gpio_get(ROT_B) ;
gpio_set_irq_enabled_with_callback(ROT_A, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL,
true, &rot_ISR);
gpio_set_irq_enabled(ROT_B, GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL, true);

```

11. One ball, one peg (Galton board physics)

Prompt:

<https://vanhunteradams.com/Pico/Galton/Collisions.html> Starting from this example, and using the pseudocode above (which comes from the collision physics for the digital Galton Board webpage), get one ball to bounce off one peg. When the ball exits the bottom of the screen, it should automatically drop again from the top. Use the default parameters provided in the image below. `peg_radius = 6`, `gravity = 0.37`, `bounciness=0.5`, `ball_radsu = 4`. for later peg spearations: vertical seption = 19, horizontal separation = 38. but for now just want one peg. maybe inside of `wallsAndEdges` function for how we know a ball has dropped on to a peg. need to add gravity. a peg is a ball with zero velociity and infinite mass. (+ the full collision-physics derivation pasted from *Collisions.html*, omitted here — it's the math derivation, not code)

Response: Fetched the actual per-frame ball-update pseudocode from the Galton lab page (the pasted content was the physics derivation, not the pseudocode). Implemented settings, `spawnBall()`, `updateBall()`, wired the ball/peg into core 0's animation thread, disabled the core 1 boid, and verified the build. **This is the original version from this turn — the user**

hand-edited it before the next turn (adding substeps, a `v_dot_n` guard, and pointer-based signatures), so this exact code no longer appears in the file.

```
#define GRAVITY      float2fix15(0.37)
#define BOUNCINESS   float2fix15(0.5)
#define BALL_RADIUS  4
#define PEG_RADIUS   6
#define PEG_VERT_SEP 19    // unused until we add more pegs
#define PEG_HORZ_SEP 38    // unused until we add more pegs
#define SCREEN_W     640
#define SCREEN_H     480

// the single peg, centered on screen
fix15 peg_x = int2fix15(320) ;
fix15 peg_y = int2fix15(240) ;

// the single ball
fix15 ball_x ;
fix15 ball_y ;
fix15 ball_vx ;
fix15 ball_vy ;
int ball_last_peg = -1 ;    // index of the last peg struck (-1 = none yet)

// Drop the ball from top-center with zero y-velocity and a small random
// x-velocity in [-0.25, 0.25) so it doesn't land on the peg dead-center
void spawnBall()
{
    ball_x = int2fix15(SCREEN_W / 2) ;
    ball_y = int2fix15(0) ;
    ball_vx = (fix15)((rand() & 0x3FFF) - 0x2000) ;    // 0x2000 = 0.25 in fix15
    ball_vy = 0 ;
    ball_last_peg = -1 ;
}

// One frame of ball physics (order matches the lab-page pseudocode)
void updateBall()
{
    // Use ball's current velocity to update its position
    ball_x = ball_x + ball_vx ;
    ball_y = ball_y + ball_vy ;

    // --- peg collision (just one peg for now, index 0) ---
    fix15 dx = ball_x - peg_x ;
    fix15 dy = ball_y - peg_y ;
    fix15 hit_dist = int2fix15(BALL_RADIUS + PEG_RADIUS) ;

    // Cheap box check first, so we only pay for sqrt when we're close
    if ((absfix15(dx) < hit_dist) && (absfix15(dy) < hit_dist)) {
        fix15 distance = float2fix15(sqrtf(fix2float15(multfix15(dx, dx) + multfix15(dy,
dy)))) ;

        // Are we in collision? (distance > 0 guards the divide below)
        if ((distance < hit_dist) && (distance > 0)) {
            // Normal vector pointing from peg to ball
```

```

    fix15 normal_x = divfix(dx, distance) ;
    fix15 normal_y = divfix(dy, distance) ;

    // -2 (n . v)
    fix15 intermediate_term = -((multfix15(normal_x, ball_vx) + multfix15(normal_y,
ball_vy)) << 1) ;

    // Teleport just outside the collision distance, along the normal
    ball_x = peg_x + multfix15(normal_x, int2fix15(PEG_RADIUS + BALL_RADIUS + 1)) ;
    ball_y = peg_y + multfix15(normal_y, int2fix15(PEG_RADIUS + BALL_RADIUS + 1)) ;

    // Reflect velocity about the normal
    ball_vx = ball_vx + multfix15(normal_x, intermediate_term) ;
    ball_vy = ball_vy + multfix15(normal_y, intermediate_term) ;

    // Only lose energy on the first strike of a new peg
    if (ball_last_peg != 0) {
        ball_vx = multfix15(BOUNCINESS, ball_vx) ;
        ball_vy = multfix15(BOUNCINESS, ball_vy) ;
        ball_last_peg = 0 ;
    }
}

// Re-spawn at the top if the ball fell out the bottom
if (ball_y > int2fix15(SCREEN_H)) spawnBall() ;

// Bounce off the top and sides (only if still heading outward)
if ((ball_y < 0) && (ball_vy < 0)) ball_vy = -ball_vy ;
if (((ball_x < 0) && (ball_vx < 0)) ||
    ((ball_x > int2fix15(SCREEN_W)) && (ball_vx > 0))) ball_vx = -ball_vx ;

// Apply gravity
ball_vy = ball_vy + GRAVITY ;
}

```

Changes to `protothread_anim` (core 0):

```

// === GALTON BOARD: drop the first ball (Claude Code assisted) ===
// (replaces the original spawnBoid for boid0)
spawnBall() ;

// === GALTON BOARD: update + draw ball and peg (Claude Code assisted) ===
// (replaces the original boid0 wallsAndEdges/fillCircle and drawArena)
updateBall() ;
fillCircle(fix2int15(peg_x), fix2int15(peg_y), PEG_RADIUS, WHITE) ;
fillCircle(fix2int15(ball_x), fix2int15(ball_y), BALL_RADIUS, color) ;
// === END GALTON BOARD DRAW =====

```

Disabling the core 1 boid in `protothread_anim1`:

```

// === GALTON BOARD (Claude Code assisted): core 1 boid disabled for

```

```
// the one-ball/one-peg checkpoint so only the Galton ball is drawn ===
// update boid's position and velocity
// wallsAndEdges(&boid1_x, &boid1_y, &boid1_vx, &boid1_vy) ;
// draw the boid at its new position
// fillCircle(fix2int15(boid1_x), fix2int15(boid1_y), 15, color);
```

12. Fixing the build errors

Prompt (pasted build output):

[ninja build errors: COLLIDE_DIST undeclared, TELEPORT_DIST undeclared, playPegSound implicit declaration, spawnBall/updateBall argument count mismatches] ive hadded changes just recently here, please debug

Response: The user had hand-edited the physics code since #11 (adding substep motion, a `v_dot_n` approach-direction guard, and pointer-based function signatures), which introduced the undefined symbols. Fixed by adding the missing macros/stub and aligning `spawnBall()`'s signature and call sites with `updateBall()`'s pointer-based one. Rebuilt and confirmed a clean build. **(current)**

```
// collision distances in fix15 (center-to-center)
#define COLLIDE_DIST  int2fix15(BALL_RADIUS + PEG_RADIUS)
#define TELEPORT_DIST int2fix15(BALL_RADIUS + PEG_RADIUS + 1)

void spawnBall(fix15* x, fix15* y, fix15* vx, fix15* vy, int* last_peg)
{
    *x = int2fix15(SCREEN_W / 2) ;
    *y = int2fix15(0) ;
    *vx = (fix15)((rand() & 0x3FFF) - 0x2000) ; // 0x2000 = 0.25 in fix15
    *vy = 0 ;
    *last_peg = -1 ;
}

// Peg-strike sound. Placeholder until the DMA sound is wired up
// (pseudocode's dma.trigger()) -- does nothing for now.
void playPegSound()
{
}
```

Call sites in `protothread_anim`:

```
spawnBall(&ball_x, &ball_y, &ball_vx, &ball_vy, &ball_last_peg) ;

updateBall(&ball_x, &ball_y, &ball_vx, &ball_vy, &ball_last_peg) ;
```

13. Fixing the double-bounce bug

Prompt:

the current code lets to double bound when the ball touch the peg and bound up an hit the invisible wall downward

Response: Diagnosed that `hitTop/hitBottom/hitLeft/hitRight` still checked against the original demo's inner box (y: 100–380, x: 100–540), not the real screen (0–640 × 0–480). Since the peg sits at `peg_y = 120`, a ball bouncing upward off the peg crossed the leftover `hitTop` line at y=100 almost immediately, causing a second phantom bounce. Replaced all four checks with direct comparisons against `SCREEN_W/SCREEN_H`. Rebuilt and confirmed a clean build.

(current)

```
// Re-spawn any ball that falls thru the bottom of the SCREEN (not the
// old demo's inner box -- hitBottom/hitTop/hitLeft/hitRight fire at
// y=380/100 and x=100/540, which cut right through where a peg near
// the top of the screen lives, causing a second phantom bounce right
// after the real peg bounce)
if (*y > int2fix15(SCREEN_H)) {
    spawnBall(x, y, vx, vy, last_peg) ;
    return ;
}

// Bounce off the screen's top/sides (only if still moving outward, so
// it can't get stuck oscillating exactly at the edge)
if ((*x < 0) && (*vx < 0)) *vx = -*vx ;
if ((*x > int2fix15(SCREEN_W)) && (*vx > 0)) *vx = -*vx ;
if ((*y < 0) && (*vy < 0)) *vy = -*vy ;
```

14. First version of this log

Prompt:

please make a log of my prompts and their according output form this chat history and write itdown as a markdown file

Response: Created this file with entries 1–13 above (text summaries only, no code attached).

15. Adding code snippets to the log

Prompt:

please also attach all code snippets that came with the outputs as well in the markdown.
do not edit the code files, but please extract from the code i ahve right now. yes i know its
not fully up to date form the repop thats okay we'll wrok on it

Response: This revision. Added every code block that was given in each response above — current, in-file snippets for entries still reflected in `animation.c` today (#10, #12, #13, and the main `main()`/display markers in #6), and the historical snippets as originally written for entries that were later replaced, reverted, or hand-edited by the user (#1, #3, #4, #5, #8, #9, #11). `animation.c` itself was not modified for this task.

16. Adding the coefficient of restitution

Prompt:

we notice that the coefficient of resitution is not included in the implementation, add that
so the ball will lose volicity as it bounce

Response: Diagnosed that the existing reflection used `intermediate_term = -2*(v_dot_n)`, the formula for a perfectly elastic bounce ($CR = 1$), with `BOUNCINESS` bolted on afterward as a scale-down of the *entire* velocity vector — including the tangential/sliding component, which the physics says a peg strike should leave untouched. Replaced it with the actual coefficient-of-restitution formula from the Collisions page ("Bouncing off a round peg, with coefficient of restitution"): $dv = -(1+CR)(n \cdot v)n$, using `BOUNCINESS` as CR directly in the normal reflection. `playPegSound()` no longer does any velocity math itself. Rebuilt and confirmed a clean build. **(current)**

```
// Only reflect if approaching; if already moving away, reflecting
// would flip the ball back into the peg (the wrong-direction bounce)
if (v_dot_n < 0) {
    // Coefficient-of-restitution bounce off a fixed peg (Collisions
    // page, "Bouncing off a round peg, with coefficient of
    // restitution"): dv = -(1+CR)(n.vi)n. CR=1 is a perfectly
    // elastic bounce (the old -2*v_dot_n); CR<1 bleeds off some of
    // the NORMAL velocity component only -- the tangential
    // (sliding) component is untouched, same as the physics.
    fix15 one_plus_cr = int2fix15(1) + BOUNCINESS ;
    fix15 intermediate_term = -multfix15(one_plus_cr, v_dot_n) ;
    *vx = *vx + multfix15(normal_x, intermediate_term) ;
    *vy = *vy + multfix15(normal_y, intermediate_term) ;

    // Did we just strike a new peg? (sound only, for now -- the
    // restitution above already accounts for the energy loss)
    if (peg != *last_peg) {
        playPegSound() ;
        *last_peg = peg ;
    }
}
```

```
}
```

17. Updating this log again

Prompt:

please make a log of all of my prompts and their according outputs and write it down in the SESSION_LOG.md in the same style that we have so far

Response: Found that this file already existed with entries 1–15 from earlier in the session. Appended entries #16 (the coefficient-of-restitution fix) and this entry, in the same Prompt/Response/code-block style, rather than recreating the file from scratch.

Codex continuation - October 7, 2026

The following entries record the prompts and responses from the available Codex conversation. Earlier entries retain their original historical attribution and code; their "current" labels describe the file at the time those entries were written, not necessarily the present implementation.

18. Resetting statistics when knob parameters change

Prompt:

can you make it so the histogram and total fallen balls resets when any parameter is changed using the knob?

Response: Inspected `animation.c` and confirmed this behavior was already implemented. Changing ball count, bounce, or gravity sets `stats_reset` only when the parameter actually changes. At the start of the next frame, before core 1 is signaled, both cores' histogram bins and fallen counters are cleared. Switching knob modes or turning past a parameter limit does not reset the statistics. No code changes were made for this prompt.

```
if (changed) stats_reset = 1 ;

if (stats_reset) {
    stats_reset = 0 ;
    memset(bins_core, 0, sizeof(bins_core)) ;
    memset(fallen_core, 0, sizeof(fallen_core)) ;
}
```

19. Checking for zero horizontal spawn velocity

Prompt:

does this code avoid the case where a ball might spawn with 0 x velocity and stay stuck bouncing on the first peg?

Response: Explained that the spawn expression forces the integer velocity to be odd and therefore nonzero. The spawn position also has approximately +/-0.5 pixels of horizontal jitter. This prevents zero horizontal velocity at spawn, but later collision damping and rounding can still reduce it to zero; it is not an absolute guarantee against every stuck-ball case. No code changes were made.

```
fix15 x = int2fix15(SCREEN_W / 2) + (fix15)((rand() & 0x7FFF) - 0x4000) ;
fix15 vx = vel2fix15(((rand() & 0x1FF) - 0x100) | 1) ;
```

20. Explaining the odd-velocity expression

Prompt:

what does it mean to force it odd

Response: Explained that bitwise OR with 1 sets the least significant bit, making the integer odd: 0 becomes 1, 4 becomes 5, 5 stays 5, and -2 becomes -1 in the representation used here. Since zero is even, the result cannot be zero. The velocity integer is measured in units of 1/1024 pixel per frame, so 1 represents a very small velocity. No code changes were made.

21. Drawing the count below each histogram bar

Prompt:

Below each histogram bar can you write the current value of each bar?

Response: Added exact bin-count labels, including zeros, beneath the histogram. Reserved space by shortening the bars and used compact 3x5 digits with alternating label rows so large counts do not overlap adjacent labels. Counts are the sum of both cores' counters. Firmware built successfully with `cmake --build build`; the UF2 was regenerated. Display behavior was not tested on hardware during this conversation.

```
#define HIST_LABEL_Y (SCREEN_H - 18) // two rows of compact count labels
#define HIST_BOTTOM (HIST_LABEL_Y - 3)
```

Histogram drawing code extracted from `animation.c` when updating this log:

```
// Compact 3x5 digits keep exact counts readable beneath the narrow bins.
// Alternate rows so even a ten-digit count cannot overlap its neighbors.
static void drawBinCount(int bin, int count)
{
    static const uint8_t digits[10][5] = {
        {7, 5, 5, 5, 7}, {2, 6, 2, 2, 7}, {7, 1, 7, 4, 7},
        {7, 1, 7, 1, 7}, {5, 5, 7, 1, 1}, {7, 4, 7, 1, 7},
        {7, 4, 7, 5, 7}, {7, 1, 1, 1, 1}, {7, 5, 7, 5, 7},
        {7, 5, 7, 1, 7}
    };
    char label[12];
    int length = snprintf(label, sizeof(label), "%d", count);
    int width = length * 4 - 1;
    int x = BINO_CENTER_X + bin * PEG_HORZ_SEP - width / 2;
    x = MIN(MAX(x, 0), SCREEN_W - width);
    int y = HIST_LABEL_Y + (bin & 1) * 8;
    for (int i = 0; i < length; i++) {
        int digit = label[i] - '0';
        for (int row = 0; row < 5; row++) {
            for (int col = 0; col < 3; col++) {
                if (digits[digit][row] & (1u << (2 - col))) {
                    drawPixel(x + i * 4 + col, y + row, WHITE);
                }
            }
        }
    }
}

// Draw the histogram under the board, scaled so the fullest bin is HIST_H tall
void drawHistogram()
{
    int bins[NUM_BINS]; // [OPT 2] sum of both cores' counts
    int max_count = 0;
    for (int k = 0; k < NUM_BINS; k++) {
        bins[k] = bins_core[0][k] + bins_core[1][k];
        if (bins[k] > max_count) max_count = bins[k];
    }
    for (int k = 0; k < NUM_BINS; k++) drawBinCount(k, bins[k]);
    if (max_count == 0) return; // zero labels remain visible after a reset

    for (int k = 0; k < NUM_BINS; k++) {
        int h = bins[k] * HIST_H / max_count;
        if (h == 0) continue;
        // bar centered on its gap, clipped to the screen at the two outer bins
        int left = BINO_CENTER_X + k * PEG_HORZ_SEP - BAR_W / 2;
        int right = left + BAR_W;
        if (left < 0) left = 0;
        if (right > SCREEN_W) right = SCREEN_W;
        fillRect(left, HIST_BOTTOM - h + 1, right - left, h, GREEN);
    }
}
```

22. Hiding balls below the peg board

Prompt:

can you stop drawing the balls when it reaches the peg board bottom so they dont cover the histogram

Response: Changed `updateAndDrawBall()` to draw a ball only while its center is at or above `BIN_LINE_Y`. Balls below the board continue updating, retain their histogram counting behavior, and respawn normally after crossing the bottom of the screen. Both cores use this shared helper. Firmware built successfully with `cmake --build build`; the UF2 was regenerated. Display behavior was not tested on hardware.

```
static inline void updateAndDrawBall(ball_t* b)
{
    updateBall(b) ;
    // Hide balls once they leave the peg board so they cannot cover the histogram.
    // Physics continues below the board until the normal bottom-of-screen respawn.
    if (BALL_POS(b->ym) <= BIN_LINE_Y) {
        drawBall(fix2int15(BALL_POS(b->xm)), fix2int15(BALL_POS(b->ym)), color) ;
    }
}
```

23. Saving the histogram-label prompt

Prompt:

can you add the drawing the numbers under the histogram prompt to a markdown file?

Response: Created `histogram_numbers_prompt.md` in the project folder with a spelling-corrected version of the request. No firmware changes were made for this prompt.

```
# Histogram count labels prompt
```

```
Below each histogram bar, can you write the current value of each bar?
```

24. Calculating mean and standard deviation from histogram counts

Prompt:

how do we calculate the mean and standard deviation of the gaussian using the text of ball counts in the histogram

Response: Explained how to treat each displayed bin count as a weight. For bins numbered $k = 0 \dots 16$, let n_k be the displayed count:

```
N = sum(n_k)
mean = sum(k * n_k) / N
standard_deviation = sqrt(sum(n_k * (k - mean)^2) / N)
```

These formulas give the mean and population standard deviation of the observed landing-bin distribution, whether or not it is perfectly Gaussian. Use counts rather than the scaled bar heights, and do not calculate when $N = 0$. For counts 10, 20, 10 at bins 7, 8, 9, the total is 40, the mean is 8, and the standard deviation is approximately 0.707 bins.

Bin centers are 38 pixels apart, with bin 0 centered at $x = 16$:

```
mean_x = 16 + 38 * mean
standard_deviation_x = 38 * standard_deviation
```

No statistics calculation or display was added to the firmware; this prompt was answered with an explanation only.

25. Updating the AI prompt session log in Downloads

Prompt:

can you go to the downloads folder and update the ai prompt session log

Response: Located `C:\Users\adl94\Downloads\SESSION_LOG.md`, read its existing prompt/response format, and appended entries 18-25 for this Codex conversation, including relevant code snippets and statistics formulas. Preserved earlier entries and distinguished the Codex continuation from the original Claude Code history. No firmware changes were made for this log update.

Claude Code continuation - October 12, 2026

26. In-depth code commenting, authors block, and prompt provenance

(Prompt spelling corrected for readability. Used in the lab report to show that Claude was also used to comment the code in depth.)

Prompt:

Hello, I need help commenting code. On top of that, it might help to also document inside the code which prompts produced certain code snippets. Please also make an authors section for Amber Li, Ivan Huang, and Zarif Karim, October 15th, 2026.

You will find prompts with their according outputs in `Lab2 - Galton/SESSION_LOG-2.md`. You may also make comments based on `Lab2 - Galton/Lab2_Galton_Notes.md`.

Then, in the SESSION LOG, also note this specific prompt, showing that Claude was also used to comment out code in depth. This prompt will be used in the lab report. Make it of similar commenting convention to my Lab 1 in `Lab1 - Synthesizing Birdsong/a_Timer_Interrupt_DDS_Demo/dactest.c`.

Here is the file to comment. It has a lot of comments already, which is good, so just add in what else you think we should add: `Lab2 - Galton/Animation_Demo/animation.c`

Response: Read `animation.c`, both logs, and the Lab 1 `dactest.c` to match its style: a header block, `// === section ===` banners, and short trailing comments on declarations and setup calls. Only comments were added; no executable code was changed. Existing comments, including the `[OPT n]` blocks, were kept. Display and build behavior were not re-tested for this prompt.

Added to `animation.c`:

- **Authors block** in the file header (Amber Li, Ivan Huang, Zarif Karim, October 15, 2026).
- **"AI-ASSISTED CODE -- PROMPT LOG"** in the header: maps each log entry (#1-#22) to the code it produced, and notes that the OPT 1-8 work is documented by the `[OPT n]` blocks and `Lab2_Galton_Notes.md` rather than by this log.
- **[PROMPT #n] tags** at the code each prompt produced, with the original request quoted: the rotary encoder state and ISR (#1-#10), the board parameters (#11), `COLLIDE_DIST/TELEPORT_DIST` (#12), the bottom-of-screen respawn (#13), `updateBall` (#11, #16), the stats reset (#18), odd-`vx` spawn (#19, #20), histogram labels (#21), hiding balls below the board (#22), and the on-screen text panel (#5, #6, #9).
- **Provenance note on #16:** the coefficient-of-restitution form was later replaced by the `-2(n.v)n` reflection plus a `bounciness` scale, and the comment says so rather than implying #16 is the current code.
- **Extra explanatory comments:** a trailing comment on every `#include`; the Q15 format and why `multfix15` widens to 64 bits; the encoder state numbering (0-3) and the full

click sequence; the packed `ball_t` fields and meta macros; the digit-glyph bit layout in `drawBinCount`; why the sound DMA stores a pointer; and the protothread, startup, and `main()` ordering.

```
// Header excerpt added to animation.c
/**
 * AUTHORS
 *   - Amber Li
 *   - Ivan Huang
 *   - Zarif Karim
 *   Date: October 15, 2026
 * ...
 * AI-ASSISTED CODE -- PROMPT LOG
 *   ... each tagged region below carries a [PROMPT #n] comment that matches
 *   entry #n in that log.
 */

// Example of a prompt-provenance tag (spawnBall)
// [PROMPT #19, #20] "does this code avoid the case where a ball might spawn
// with 0 x velocity and stay stuck bouncing on the first peg?" / "what does it
// mean to force it odd" -> OR with 1 sets the low bit, so vx can never be 0.
```