

Ping Pong Success Detector

1st Sophie Liu

6-3

MIT

Cambridge, USA

sophie05@mit.edu

2nd Justin Buonato

6-1

MIT

Cambridge, USA

buonato@mit.edu

Abstract—We present Ping Pong Success Detector, an FPGA project that uses audio and video analysis to detect when it bounces and determine whether the player successfully returns the ball. We developed a custom color space to easily mask for orange, edited the camera configuration to increase the framerate, and convolute the audio to isolate bounce events. Additionally, we implemented a height estimation module which involved Bresenham stepping and a CORDIC SQRT IP module, though this approach proved unreliable in the real world of noisy pixels.

Index Terms—Digital systems, Field programmable gate arrays, IP, CORDIC, Bresenham, Color spaces.

I. INTRODUCTION

Our project tracks a ping-pong ball, detects when it hits the table, and informs users when they fail a rally. This is harder than it sounds: estimating a 3D trajectory from 2D imaging alone is nontrivial. Furthermore, our camera's framerate introduces more uncertainty, as a ping-pong ball can travel half a meter in a single frame, making precise localization a challenge.

II. BOUNCE LOCATION

We determine the bounce location by combining an audio-detected bounce time with frame-by-frame center-of-mass tracking and orange color masking. In particular, when the FPGA detects the sound of a ping pong ball bounce, it guesses at the bounce location by linearly interpolating between the ping pong ball's location in the previous frame and its location in the next frame, weighted by when the bounce sound occurred.

A. Mask

To isolate the orange ping pong ball, we developed another color space by applying a rotation matrix to an initial YCrCb transformation, creating a "Chroma Orange" axis we can easily threshold on. This module only takes two clock cycles, making it quite efficient.

$$\begin{bmatrix} Y \\ Co \\ Cp \end{bmatrix} = \frac{1}{1024} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos 45^\circ & -\sin 45^\circ \\ 0 & \sin 45^\circ & \cos 45^\circ \end{bmatrix} \begin{bmatrix} 306 & 601 & 116 \\ 512 & -429 & -83 \\ -173 & -339 & 512 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1)$$

Color Planes Compared at Y=0.6, R,G,B from 0 to 1

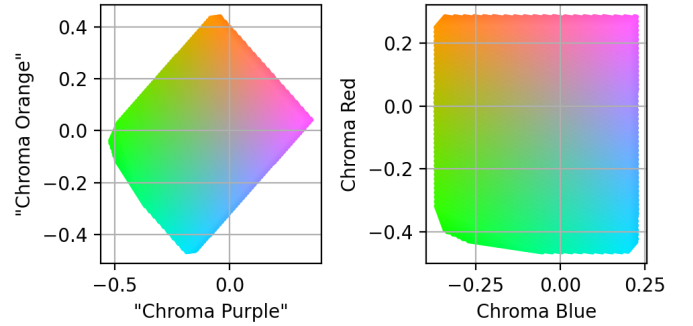


Fig. 1. Visualizing the Matrix Transformation into our color space with Orange as the vertical axis.

B. Camera Configuration

Due to the uncertainty in position created by motion blur in our images and our tolerance for lower image fidelity in this scenario, we set exposure to the lowest value possible given lighting conditions, modifying the camera's 0x3500-0x3503 registers. We also set the PLL to double the frame rate from the initial configuration.

C. Microphone

We interface with an INMP441 microphone through our I2S controller and receiver module, providing 24-bit audio at 31250 Hz into an audio buffer that takes its absolute value and truncates it to 16 bits. From experimentally recording matches of ping pong, we observed that the ball hitting the table was so loud and sharp compared to footsteps that it did not necessitate much signal processing to capture. We implemented a $[-1, \dots, -1, 1, \dots, 1]$ kernel of length 256 with a running sum that we modify each cycle. This process is particularly efficient with 3 additions per cycle as we subtract the new sample, add twice to shift the sample from -1 to 1, and subtract the oldest sample. This kernel shape was chosen as it matches the jump in volume from background noise to the ping of the ball. The length was experimentally chosen by offloading audio from the FPGA onto a computer via UART and seeing that in software implementations, starting around this length, there is only one large spike in response.

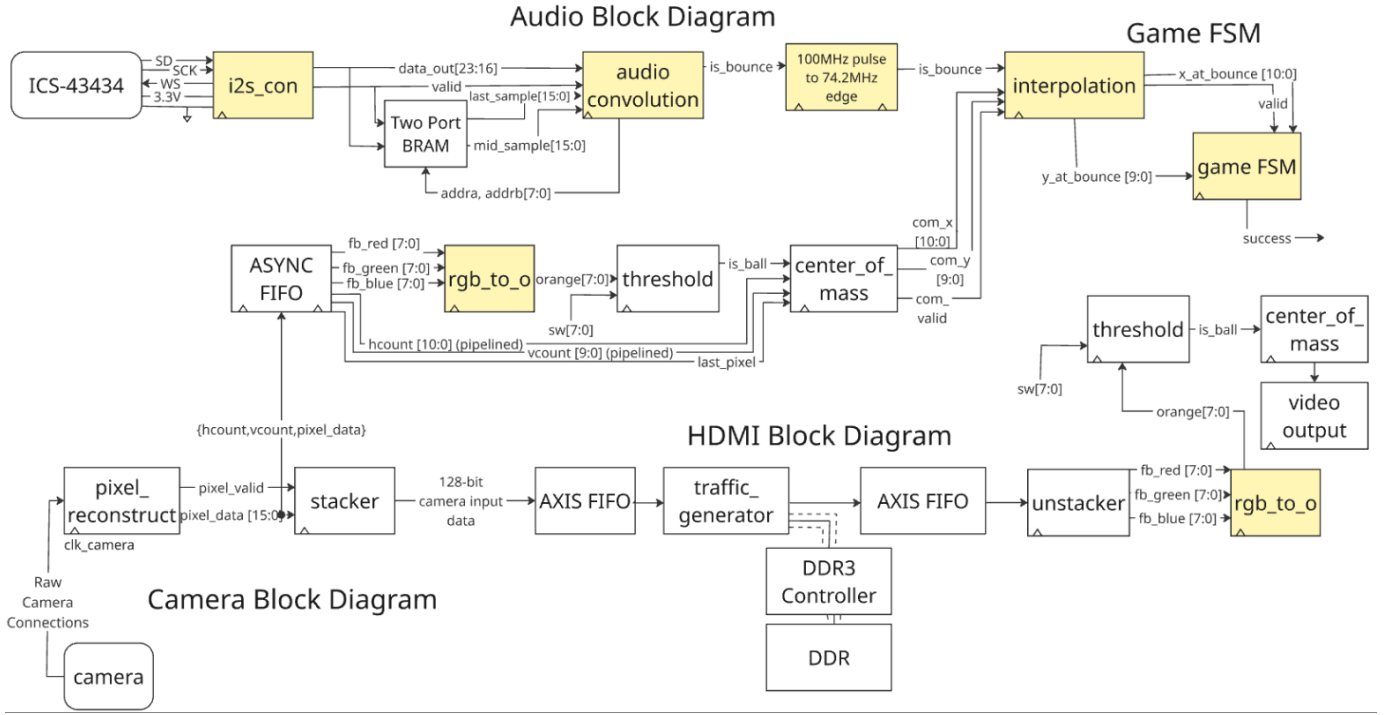


Fig. 2. Block Diagram of our entire system.

D. Interpolation

In order to determine where the ping pong ball bounces, we combine the "when" information we gained from the audio analysis with the "where" information given by the camera and masking. In particular, by interpolating the location of the ball from the previous frame and the next frame based on the number of clock cycles that passed between the previous frame and the ball bouncing, we get an approximate position of the ball at the time of the bounce. Our interpolation module has a latency of 4 clock cycles after the next center of mass is inputted.

In order to wire interpolation effectively, however, we need the inputs of interpolation to correspond to the camera's h_count and v_count , rather than the HDMI video's h_count and v_count . After all, HDMI video exhibits repeated frames and frame tearing, which would render our interpolation calculations quite inaccurate. Thus, we incorporated an ASYNC_FIFO into our design to facilitate crossing from the 200 MHz camera clock domain to the 74.25 Mhz pixel clock domain.

III. LOCATION TRACKING: HEIGHT

The x and y locations of the ping pong ball can simply be estimated as the x and y coordinates of the center of mass of the ping pong ball in the video feed. Determining the z location of the ping pong ball, however, is more difficult.

A. Relative Sizes

We implemented a module which estimates the height of the ball based on its relative diameter. More specifically, we

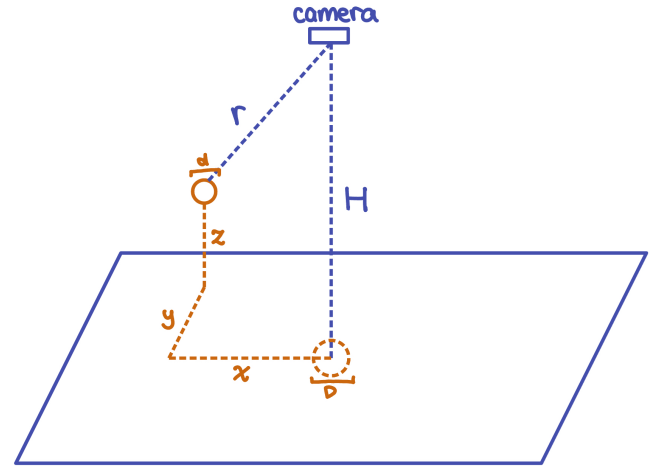


Fig. 3. The physical setup of the system, with some variables labeled that are useful for calculating height. D is the perceived diameter of the ball when the ball is placed at the center of the ping pong table.

estimate that the height is roughly

$$z = H - \sqrt{\frac{H^2 D^2}{d^2} - (x^2 + y^2)}. \quad (2)$$

In order to implement this on our FPGA, we use a width-32 CORDIC SQRT IP block.

B. Diameter Detection

Determining the perceived diameter of the ball in a given frame posed its own difficulties, however. In order to maximize

accuracy and reduce the effect of motion blur, we implemented a module to measure the diameter of the ball in the direction perpendicular to the motion. To do so, we use a Bresenham line stepping implementation that starts at the center of the ball and draws two roughly straight rays, perpendicular to the ball's motion, from the center of the ball. Bresenham's line algorithm is a classic rasterization technique that approximates a straight line between two points on a pixellated screen using only integer arithmetic. The cheap cost of Bresenham makes it extremely well-suited for FPGA implementations. Intuitively, Bresenham's algorithm does so by stepping in one direction, either x or y , until the error in the other direction becomes too large.

In order to actually use the Bresenham line stepping algorithm, however, we need to be able to go backwards and upwards through the raster stream, so we filled a 256x256 binary buffer in the BRAM with the bounding box around the ball.

C. Flaws in Calculating Height

Though our height and diameter modules passed all our testbenches with flying colors, I failed to consider one thing when designing these beautiful, complex, elaborate modules: the world is not a testbench. After we deployed them to the FPGA board, we realized that, though the orange masking works extremely well when the ball is stationary, the moment it begins to move, the edges of the ball are lost in noise. Though we attempted various fixes—taking the largest diameter rather than the smallest, for example, or attempting to smooth out the inputs to generate a smoother output—the noise inherent to the camera and our method of tracking the ball made this method simply infeasible.

Additionally, I found that my method of calculating height was not the most accurate. Rather, it is possible, with just the x and y frame locations of a ball and the diameter of the ball, to determine how high the ball is. This second method capitalizes on the fact that every point on a line emanating from the camera is at the same location in the camera frame. Then, by utilizing similar triangles, one can prove that the height z of the ball off the table is

$$z = H \left(1 - \frac{r}{d\sqrt{H^2 + x^2 + y^2}} \right).$$

Even though this second method is extremely accurate, it would also be even more negatively impacted by noisy diameter values than the initial estimation I started with, as the derivative of d with respect to z is larger in the second case than in the first. Even so, if I were to work more on this project, I would still want to focus on exploring other potential methods for determining ball height, or attempt to refine our ball tracking method in order to make these ball height algorithms actually feasible.

IV. EVALUATION

A. Latency and Throughput

Our system processes two independent high-rate streams: a 200 MHz camera input and a 100 MHz audio sample input. As

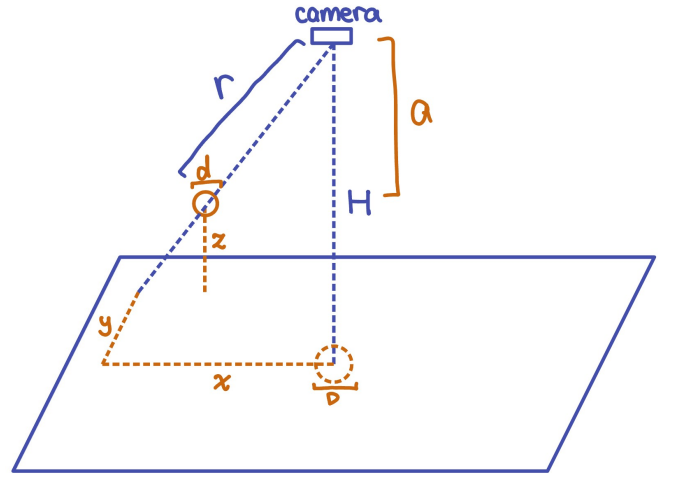


Fig. 4. Another method for calculating height.

such, most of our computational modules were designed with fixed, short latencies in order to sustain real-time throughput under tight timing constraints:

- Our rgb to orange module performs at 2-cycle latency at 200 MHz.
- Our interpolation module, as mentioned above, has a 4-cycle latency after receiving the next frame's center of mass. 4 cycles are entirely negligible compared to the frame period of the camera. As the interpolation module only does at most one interpolation per frame, this 4-cycle latency is more than small enough.
- Our audio convolution module has a throughput of 1 sample per clock on the 100 MHz clock cycle. Its latency, however, is higher, as it equals the depth of the 256-sample window that gets stored in the BRAM. A latency of 256 samples corresponds to roughly 8 ms of 31250 Hz audio. As audio convolution is only required to be much smaller than the length of a frame in order to successfully provide a good bounce time, this latency is entirely acceptable.

We do intentionally introduce two throughput overheads, as well: first, we use an asynchronous FIFO in order to cross clock-domain boundaries between camera, which runs on a 200 MHz clock; video, which runs on a 74.8 MHz clock; and audio, which runs on a 100 MHz clock. This required us to add some pipelining, but it greatly improves correctness. The second overhead we attempted to introduce was my BRAM-based diameter buffering.

B. Resource Utilization

We used 38.5 of our 75 BRAMs, mainly in the FIFOs; 7 of our 120 DSPs, mainly in the camera conversion block; 4294 of our 32600 LUTs, and 4269 of our 65200 registers. Overall, our resource utilization profile highlights the fact that our design is relatively compute-light and memory-heavy, since BRAM was by far our most restricted resource.

C. Timing Requirements

Our post route timing summary actually reports the extremely concerning fact that our WNS is -3.668 ns. However, further investigation reveals that this slack violation occurs when crossing from our 200 MHz camera clock cycle to our 74.25 MHz pixel clock cycle and is, in fact, an entirely expected and intended behavior. In fact, this boundary is intentionally asynchronous and handled appropriately through an asynchronous FIFO, so it's not a timing error. Aside from the slack that arises from crossing clock domains, all paths within synchronous clock domains meet timing. Our design satisfies all meaningful timing requirements.

D. Sources of Error

With a project such as ours focused on interpreting, analyzing, and interacting with signals in the real world, an important part of evaluating our design then becomes determining how well it stands up to natural sources of error. For instance, lighting variation and sensor noise caused the edge of the ball to be instable, which greatly impacted our module's accuracy and usefulness. Similarly, environmental noise and even the angle of our microphone were able to negatively impact our audio analysis. In fact, our audio convolution occasionally produced clear spikes representing bounces when there was no bounce anywhere on the premises.

When designing future hardware projects, I'll definitely remember to tailor my designs to being noise-resistant. With that said, the core functionality of our project, tracking ball location and reacting to bounces, performed reliably and accurately.

E. Use Cases

Our system successfully performs real-time ball tracking, bounce detection, and bounce location interpolation, as desired. In fact, we even wired up a `ball_sprite` module so that at each bounce, an image of the ball would appear at the location the ball was just occupying. We did not manage our reach goal of successfully estimating ball height, however.

Other potential use cases for our project include trajectory prediction, training feedback, and automated scoring, all of which could be integrated into our design without changing its structure overtly. Additionally, we could also pursue the other features we considered but did not implement, including the table boundary detection module that we discussed in our last report.

V. RETROSPECTIVE

In hindsight, we learned a few rather important lessons during the development process: Firstly, hardware engineering should not be approached with the same mindset as software engineering. I designed, implemented, and tested my diameter and height calculators in a world of perfect inputs that didn't remotely reflect the actual data my modules were given after I deployed them. I knew that I would likely need to contend with unforeseen bugs after porting them to the FPGA, but I didn't

even consider that my entire approach might not be particularly feasible as matters—and our ball-tracking mechanism—stood.

On the topic of my diameter and height calculators, another important lesson I learned was to focus on the minimal viable product before working on anything fancy. Because I believed that setting up top-level wiring would be simple, I spent the majority of the last two weeks of the project working on the height and diameter calculators rather than working on the top-level wiring, which ended up stressing me out a lot.

Another important lesson we learned was the value of parallelizing work. We implemented and tested all our modules separately, allowing us to come together and fit everything together (almost) seamlessly in very little time. As a corollary, however, when Justin's laptop's could no longer connect to WiFi and we could no longer work in parallel, our efficiency diminished drastically.

This is said, however, with the caveat that parallelization does require communication: in particular, because we implemented, wired, and tested the video and audio parts separately, we ended up accidentally using two different clock cycles for the video and audio, thus requiring us to cross clock domains and increasing the complexity of the process where complexity was not needed.

On a more technical side, one major skill I learned was how to cross clock domains. In previous labs, all of the clock domain crossing had been black boxed away for us in clock wizards and provided code, so implementing clock domain crossing this time was very educational.

VI. CODE REPOSITORY

Our code can be found at the following repository: <https://github.mit.edu/6205F25/fa25-6205-team13>.

VII. INDIVIDUAL CONTRIBUTIONS

A. Code

Sophie wrote most of the computation-heavy modules, including the interpolation module, the game FSM, and the, sadly unsuccessful, diameter detection and height calculation modules.

Justin worked on interfacing with the microphone, writing the audio convolution module and the i2s controller, as well as experimenting with audio to determine what the best method for detecting ping pong bounces would be.

Sophie and Justin both worked on wiring the top-level module.

B. Report

Sophie and Justin worked jointly on the Block Diagram and the Preliminary Report. Justin updated the block diagram for the Final Report. Sophie wrote the rest of the paper that wasn't already in the Preliminary Report, as well as the Code Provenance document.

C. Acknowledgements

Thank you to Joe Steinmeyer for your help and guidance; we really loved this class! And thank you to Kiran Vuksanaj for your help with cameras!