La PC-na

Final Project Report
6.111 Fall 2016

Matt Basile and Zareen Choudhury
December 16, 2016

Table of Contents

1 Introduction… ...2
2 Technical Overview… ...2
3 External Components… ...3
    3.1 Cue Position Tracking (Zareen Choudhury)… ...4
        3.1.1 Tracking Algorithm… ...5
        3.1.2 Cue Hardware… ...5
    3.2 Cue Speed Calculation (Zareen Choudhury)… ...6
        3.2.1 Speed Calculation with Accelerometer… ...6
        3.2.2 Speed Calculation with Cue Tracking… ...8
    3.3 Calibration (Zareen Choudhury)… ...9
4 Internal Components… ...11
    4.1 Ball Position Updates… ...11
        4.1.1 Wall Collision Checker (Matt Basile)... ...11
        4.1.2 Ball Collision Checker (Matt Basile)... ...11
        4.1.3 Cue Collision Checker (Zareen Choudhury)... ...13
        4.1.4 Pocket Checker (Matt Basile)... ...13
        4.1.5 Position Handler (Matt Basile)... ...13
        4.1.6 Friction (Zareen Choudhury)… ...14
    4.2 Game FSM (Zareen Choudhury)... ...15
    4.3 Game Visuals (Matt Basile)... ...17
5 Lessons Learned… ...17
    5.1 Matt Basile… ...17
    5.2 Zareen Choudhury… ...18
6 Future Work & Conclusion… ...19
7 Acknowledgements… ...19

1 Introduction

The luxury of purchasing separate pool tables, foosball tables, and air hockey tables is beyond the budget of most individuals, particularly college students. We propose a more cost effective solution that synthesizes a multitude of tabletop entertainment options into a single system. Our project idea is based on the concept of combining virtual and physical interactions for a new entertainment experience.

In our project, La PC-na, we implement one of these classic tabletop games -- pool -- by using physical cues in conjunction with virtual balls and pockets. The pool table is virtually displayed on a screen. Players use a physical cue to "hit" the virtual balls, and the projected game updates in real-time to mimic the physics of pool table collisions. With our project, we seek to combine the appeal of classic games with the hands-on, real-time interaction that today's users have come to expect.

2 Technical Overview

The virtual pool table, balls, and pockets were displayed on a TV screen that was oriented horizontally on a tabletop. An NTSC camera with an infrared (IR) filter was mounted above the TV screen to capture real-world motion, as shown in Figure 2.1. IR LED's were mounted on the physical cue so that the camera could track the position of the cue. All processing was performed on the 6.111 Labkit.

Figure 2.1: Hardware setup with NTSC camera mounted above TV screen.

We divided our implementation into external components and internal components. External components included cue position tracking (Section 3.1), cue speed calculation (Section 3.2), and calibration (Section 3.3). Internal components included ball collision checkers (Section 4.1.1-4.1.4), ball position handler (Section 4.1.5), friction (Section 4.1.6), and the game FSM (Section 4.2). Zareen Choudhury primarily worked on external components, and Matt Basile primarily worked on internal components. A high-level block diagram of the system is depicted in Figure 2.2.

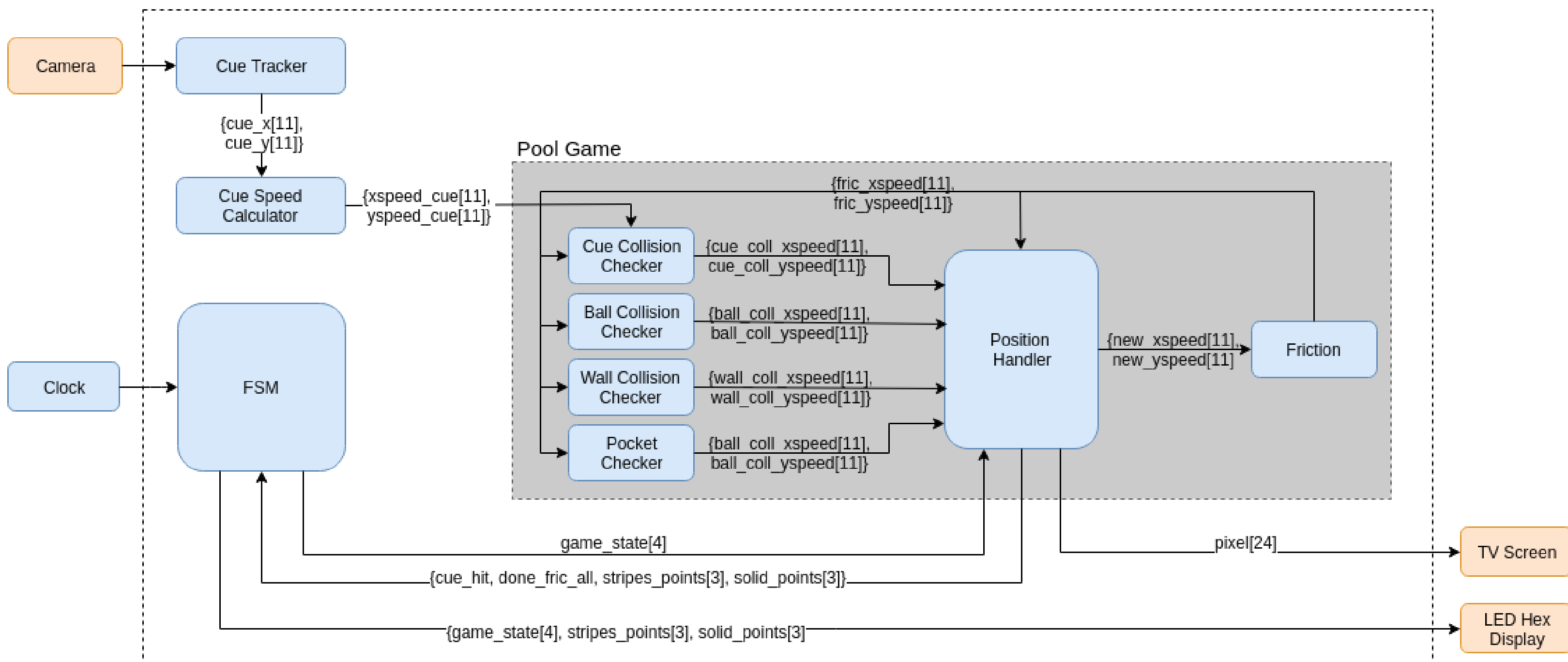


Figure 2.2: Block diagram of the system.

In the first step of the block diagram, the NTSC camera data was consumed by the Cue Tracker module to determine the x- and y-position of the cue. These positions were given to the Cue Speed Calculator, which calculated the speed of the cue's movement. The positions and speeds were inputs to the Pool Game module. Within the Pool Game module were all the collision checker submodules (Cue Collision Checker, Ball Collision Checker, Wall Collision Checker, Pocket Checker), the Position Handler, and the Friction module. All updates on ball position and velocity were done internally within Pool Game. Finally, the game FSM received signals from Pool Game indicating when to transition states. The outputs of the Game FSM were used to update scores on the Labkit's LED hex display, and the pixel output of Pool Game was used to update the image on the TV screen over VGA.

3 External Components

The external components consisted of a TV for displaying the virtual pool table, physical cue stick to interact with the virtual pool table, and NTSC camera for tracking the cue stick. To track the cue, infrared (IR) LED's were mounted on the stick, and the NTSC camera was covered with an IR filter (floppy disk film) to detect the LED's.

3.1 Cue Position Tracking (Zareen Choudhury)

3.1.1 Tracking Algorithm

I iterated through two high level algorithms for tracking the IR LED's on the cue stick. The first algorithm could track only a single LED, while the second could track multiple LED's. My first pass algorithm followed a center of mass approach and was written on top of the sample NTSC camera code provided on the 6.111 course website (NTSC Camera Sample Verilog under "Tools"). The provided sample code decodes the video stream output of the NTSC camera, saves it in ZBT memory, and reads the pixel values from ZBT to display the camera image over VGA. My first cue tracking algorithm looked for pixel intensities higher than a specified threshold (128). Once a pixel satisfying this condition was found, its hcount and vcount were stored as the pixel's corresponding x- and y-position. The average x- and y-values of all the pixels satisfying this condition represented the center of mass of the LED. This algorithm was tested with a single diode in a TV remote, and produced fairly accurate real-time tracking.

For our application, we would have to track multiple LED's since the cue would be mounted with LED's at both the front and back of the stick. Averaging across all bright pixels would not work, as there were multiple regions of interest. Instead, I followed a different approach that formed populated various regions of interest. This algorithm is illustrated in Figure 3.1. The building block of this algorithm is a "zone." The first zone is initialized around the first bright pixel found, using the pixel as its center coordinates. When another bright pixel is found, the algorithm checks if it lies within specified margins around the first zone. If it does, the first zone adjusts accordingly to accommodate the new pixel. Otherwise, if it lies outside the first zone's bounds, a second zone is initialized around the new pixel. Each zone cascades a signal to the subsequent zones indicating whether the pixel has been placed in a zone, or whether the algorithm should continue searching.

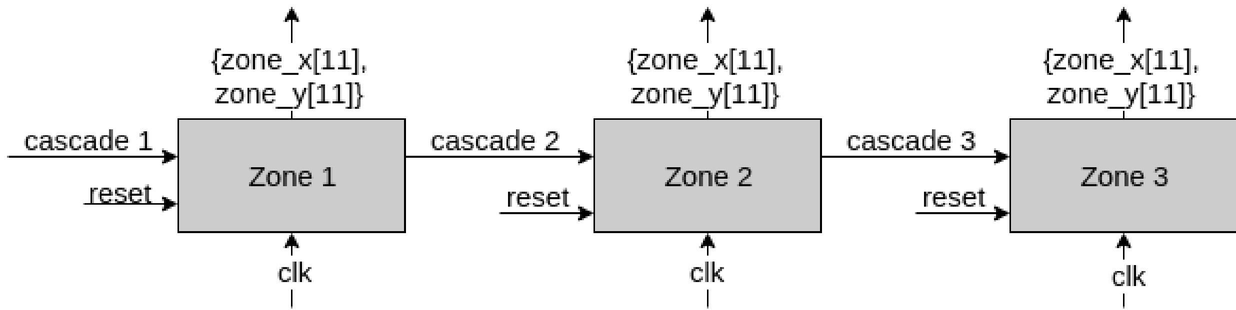


Figure 3.1: Block diagram of multiple zone tracking algorithm.

This algorithm was tested by mounting LED's on multiple locations on the cue and visualizing the tracked zones with sprites (Figure 3.2). Through visual inspection of the sprites' movements, we found that the algorithm successfully and accurately tracked multiple LED's in real time.

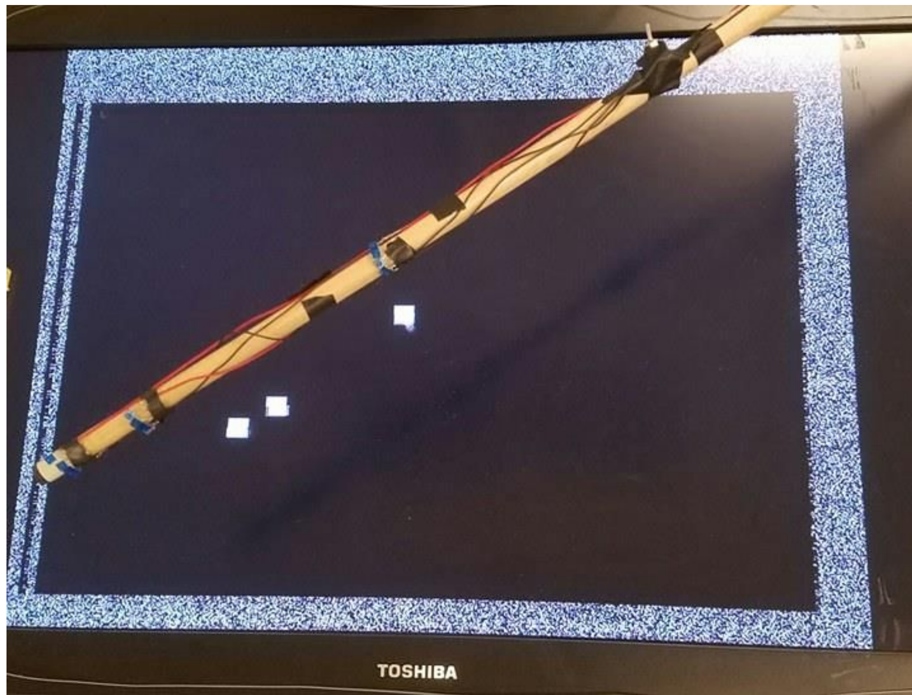


Figure 3.2: Multiple IR LED's tracked with NTSC camera.

3.1.2 Cue Hardware

The configuration of the LED's mounted on the cue had to be revised a few times to optimize tracking. In the first version of the configuration, there were two rings of LED's: one at the front of the stick and one at the back (Figure 3.3a). Two rings were used in order to find the vector of the cue's position. Four LED's were used for each ring, spaced out evenly over the circumference of the stick. The LED's were spaced out so that the cue could be detected regardless of its angle of rotation.

However, when this setup was tested through visual inspection, I found that the LED's appeared very faint. I tested increasing the voltage across each LED by arranging them in parallel, decreasing the resistance, and increasing the supply voltage. Even with these changes, the LED's continued to appear dim.

Figure 3.3: (a) First cue LED configuration with evenly spaced diodes.
(b) Second cue LED configuration with diodes angled upwards.

Upon further testing, I discovered that what mattered more than the voltage drop was the angle of the diodes. If the diodes faced directly upwards at the camera, they appeared very bright, but at even a slight angle away from the camera, they appeared substantially dimmer. Therefore, I revised the configuration so that the LED's faced directly upwards towards the camera, and lined them up right next to each other without any spacing, in order to increase the perceived brightness (Figure 3.3b). This configuration provided much better and more consistent tracking.

3.2 Cue Speed Calculation (Zareen Choudhury)

In our initial design, we planned to determine the speed of the cue from the MPU-9250's 3-axis accelerometer. I first attempted to calculate speed from accelerometer data, which is described in Section 3.2.1. However, due to noisy data, I decided to switch to calculating speed from tracked cue positions instead. This is described in Section 3.2.2 and is what we used in the final configuration.

3.2.1 Speed Calculation with Accelerometer

I used an MPU-9250 provided by Joe Steinmeyer, and used the SPI serial protocol to transmit data to and from its 3-axis accelerometer. The master in this configuration was the FPGA, and

the slave was the MPU-9250. The accelerometer data was read using an FSM that consisted of an address state, a data transfer state, and an idle state (Figure 3.4).

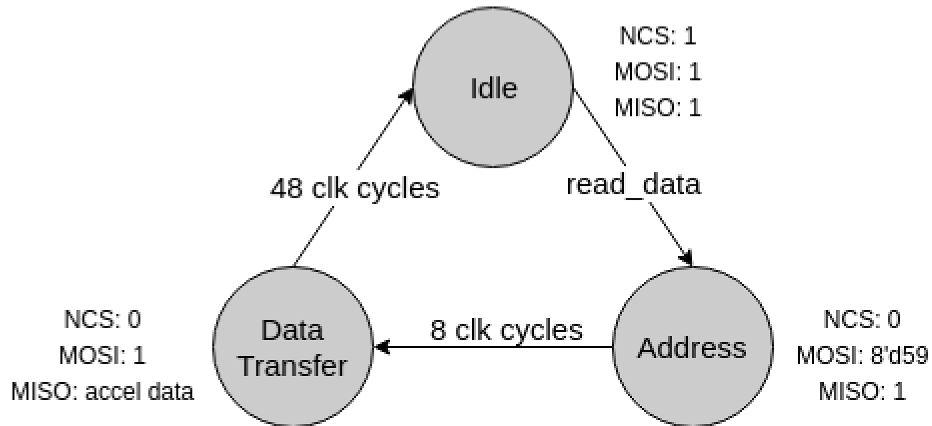


Figure 3.4: FSM of accelerometer reader module.

In the address state, the master serially sent the slave the register addresses from which accelerometer data would be read in the data transfer state. Although there were six registers containing all the data (two registers for each of the three axes), only the first address needed to be provided (Register 59). After this 8-bit register address was sent over MOSI, the FSM transitioned to the data transfer state. In data transfer, the data was sent serially from the slave to the master over MISO, from most significant to least significant bit, in order of x, y, and z. Each axis consisted of 16 bits of data for a total of 48 bits. Once all the data was transferred, the FSM transitioned to the idle state until prompted to read data again. The chip select pin was active low during addressing and data transfer, but high during the idle state. The clock provided by the master (SCK) was a 1 MHz clock.

The accelerometer readings were tested by rotating and moving the chip and observing corresponding oscilloscope outputs. Sample oscilloscope outputs are shown in Figure 3.5. While the output values were overall consistent with expectations, the data was still noisy. To reduce noise, I implemented a low-pass filter that returned a rolling average of accelerometer values over the past 8 clock cycles. The average data was converted into a cue speed by finding the magnitude of the vector formed by the 3 axes and assigning a pixel speed based on experimentally determined thresholds.

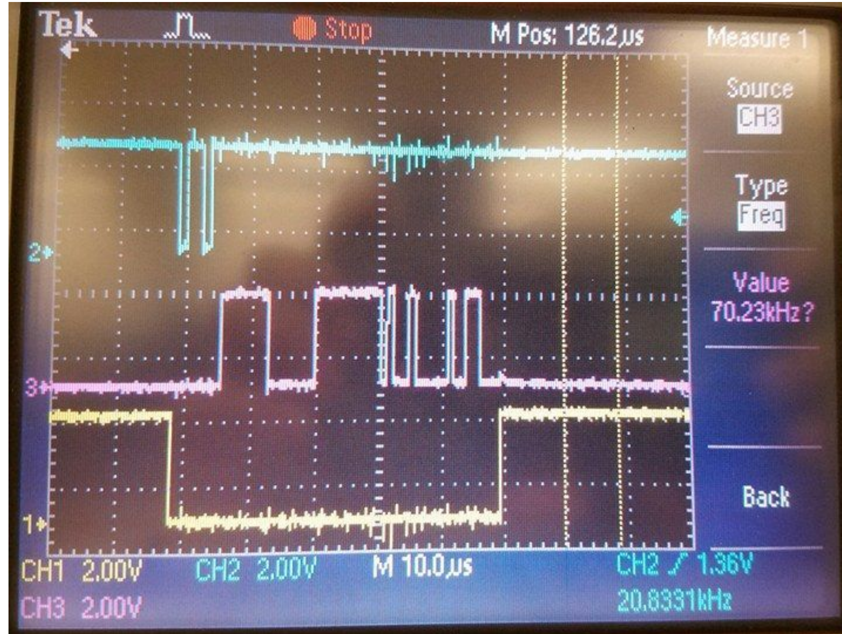


Figure 3.5: Oscilloscope output of accelerometer data.
Blue is MOSI, purple is MISO, and yellow is NCS (chip select).

Unfortunately, even with the low-pass filter, the data was still fairly noisy for our purposes. When tested with simply keeping the chip at rest, the cue speed switched threshold ranges approximately 2 out of 10 trials. Given that the noise of the accelerometer would compound with imprecisions in the tracked cue position and negatively impact the game experience, I decided to try to calculate cue speed from the tracked position instead. My reasoning was that a single external source of noise would reduce overall impact of imprecisions.

3.2.2 Speed Calculation with Cue Tracking

The cue speed was calculated from the tracked cue position by maintaining a circular buffer containing the cue front's last eight x- and y-positions. The difference in cue position divided by the number of clock cycles gave an approximation of the cue speed. This value was converted into an appropriate pixel speed based on experimentally determined thresholds. This tracking-based cue speed module was tested by outputting the calculated position differences on the LED hex displays. The cue was moved from set starting positions to set ending positions, and I checked whether the displayed numbers matched expectations. The results indicated that the calculated cue speeds were quite accurate and close to expectations. Given that this was a fairly simple algorithm that produced accurate, real-time speeds while reducing the number external components, I decided this was a better approach compared to the accelerometer, and ended up using it for final integration.

3.3 Calibration (Zareen Choudhury)

The purpose of calibration was to align the physical cue with the virtual setup as closely as possible, in order to provide a more realistic game experience. One of the complications with this was that the resolution of the NTSC camera was 525x480, but the game screen had a resolution of 1024x768. As a result, the camera could only track the cue in a small portion of the game screen, as illustrated in Figure 3.6. In the figure, the black rectangle is the camera's field of vision, while the gray represents the full extent of the game screen.

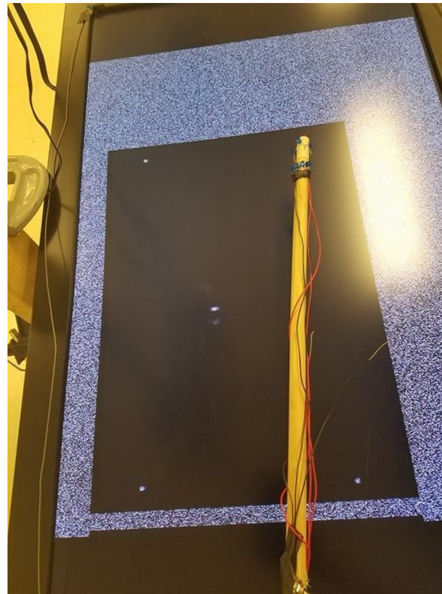


Figure 3.6: Camera bounds (black) versus TV screen bounds (gray).

My first approach to solve this problem was to stretch the camera's coordinates to match the resolution of the game screen. To do this, I fixed LED's to the four corners of the TV screen (Figure 3.7). I implemented a calibration module that stored the coordinates of the TV screen's corners by tracking the four LED's. These coordinates were then used to find the scaling factor between the camera and the game screen.

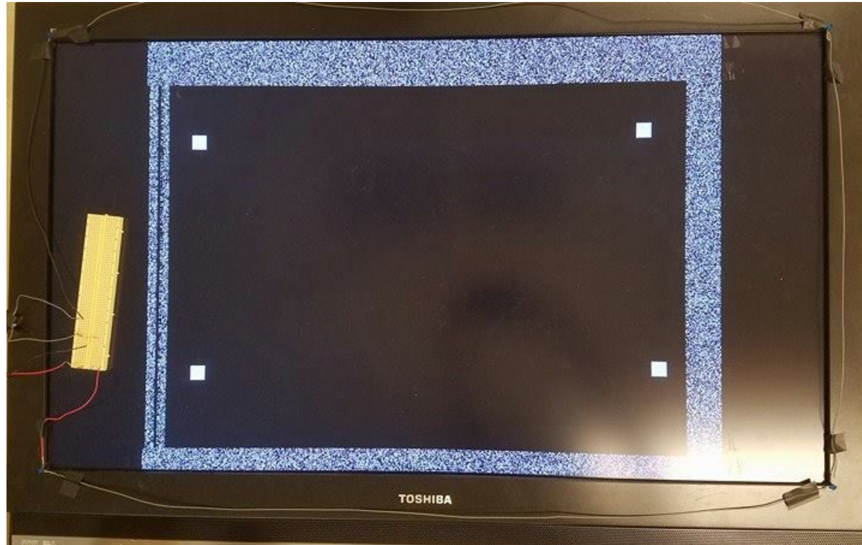


Figure 3.7: LED’s mounted on TV corners for calibration.

This approach allowed the cue to be tracked over the full bounds of the game screen, as opposed to a fraction of it. However, the virtual and physical cue were not aligned because the physical cue’s coordinates were stretched onto a larger coordinate system. My next approach to this problem was to instead decrease the resolution of the game screen. This way, the camera’s field of vision would appear to occupy a larger fraction of the game screen.

I tested decreasing the resolution from 1024x768 to both 800x600 and 640x480. The XVGA module was adjusted accordingly for each resolution, using the VGA Timings table on the 6.111 course website (“VGA Video Output” under “Labkit”). For the 800x600 resolution, a 40 MHz pixel clock was used, and for the 640x480 resolution, a 25 MHz pixel clock was used. While the camera’s apparent field of vision successfully increased and the physical cue aligned better with the virtual cue, I discovered that the slower pixel clock decreased the speed of tracking. The lag in cue tracking visibly increased, causing the virtual cue to jump around sporadically. There was a direct tradeoff between increasing the size of of the camera’s tracking bounds and maintaining real-time tracking. Ultimately, we chose to use the middle resolution, 800x600. The 40 MHz clock still enabled reasonably fast and smooth cue tracking, and the camera was able to track about 50% of the game screen. While the tracking area was not optimal, having real-time cue response was more important for our game experience.

4 Internal Components

4.1 Ball Position Updates

4.1.1 Wall Collision Checker (Matt Basile)

The wall collision checker is the simplest collision detection module. By nature of the shape of the balls and table, the pool balls could only collide with the walls at either 0, 90, 180, or 270 degree angles. Therefore, when checking for wall collisions, we only needed to check those four points on the circle's radius, with a simple series of less than and greater than conditionals. A single wall collision checker module detected collisions between a single ball and all four walls, and output a signal depending on whether the vertical or horizontal walls were hit. To update the speeds, the sign of the appropriate vector was simply reversed.

Occasionally, balls would jump past the walls and continue off the screen, especially if they were in a series of rapid collisions with other balls. This occurred because the speed of the ball became so great that it jumped past the wall and thus escaped detection. Implementing a maximum speed of less than the radius of the ball prevented this by keeping the ball from jumping greater than its radius in length in one move. Another solution was to ensure the ball only collides with the wall when moving in the appropriate direction (i.e. the ball must bounce left after hitting the right wall). This helped remove a fair amount of our glitchy wall collisions by keeping balls from bouncing off the 'other side' of the wall and wrapping around the screen.

4.1.2 Ball Collision Checker (Matt Basile)

The ball collision checker was an individual module which took in two balls' positions and speeds as input and constantly checked for a possible collision. If a collision was detected, the Ball Collision Checker calculated the new speeds of both balls according to their impact speeds and positions, and then passed them along to the position handler.

To create realistic collisions, we modeled ball hits as 2-D elastic collisions, meaning there is no energy transferred between collisions. In such a collision, calculating the resulting movement vectors is fairly simple. Simply calculate the normal and tangential vectors for each ball at the point of collision, swap the normal vectors between the two balls, and recalculate the ball's speed with its new normal and old tangential vectors.

One priority we had when creating the Ball Collision Checker was to keep its calculations as lean as possible. The number of Ball Collision Checker's scaled at $O(n^2)$ with the number of balls *n;* thus we wanted to ensure that as little processing power was needed for each single Ball Collision Checker. The primary way we achieved this was by avoiding both square root and division operations (except division by powers of 2). However, this made our physics

calculations rather difficult, as they relied on calculating the normal and tangential vectors given each ball's position and current speeds, and trigonometry relies on square root and division. A few key realizations helped us simplify the collision calculations into a few steps.

1. We could calculate the unit normal vector for each ball by drawing the normal vector between the two ball's centers and then dividing by the magnitude which was fixed at two times the radius of the balls (16*2=32), which was a power of 2.
2. The tangential and normal vectors of both balls were related by sign swaps for each x and y coordinate. This meant we could calculate both the normal and tangential vectors of each ball with minimal computation.
3. Once we calculated the tangential vectors, we swapped the normal vectors of each ball.
4. Since the normal and tangential vectors were also calculated by their x and y components, we could recalculate the new direction and speed for each ball by simply adding the x components of its new normal and tangential vectors, and then adding the y components.
5. These calculations could be done in a single vsync cycle, allowing us to maintain timing with the rest of the system.

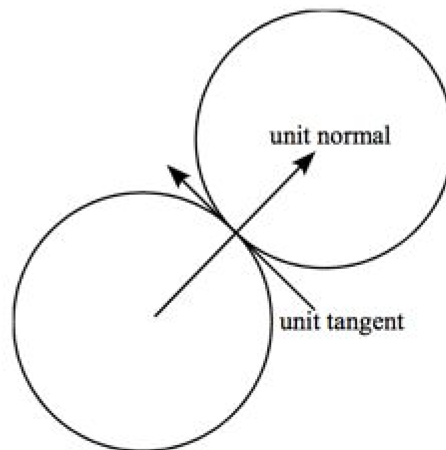


Figure 4.1: *Diagram of the necessary vectors in elastic collision calculation.*

One way we attempted to make ball collisions feel more realistic was by looking ahead when detecting collisions. We added the current speed to the current ball position when looking for collisions, since the nature of blocking assignments ensured that we had to wait 1 clock cycle before the ball position actually updated. A problem we ran into was in our substitution of right shifts for division. Right shifts do not store a remainder, and they round down. Thus, collisions that happened at too shallow of an angle were not calculated properly, and the resulting ball would move at a 0 or 90 degree angle. One possible fix would be to either implement a rudimentary decimal and remainder system, or to simply scale up the speeds when doing calculations (5 -> 500) so Verilog would always be operating in whole numbers but we would have more significant digits when doing calculations.

Much like the wall collisions, we also ran into the problem of balls moving too quickly and thus

skipping over some or all of the other balls, causing collisions to appear messy or unrealistic. Unfortunately, the same fix of applying a maximum speed equal to the radius of the ball did not work as effectively because even a ball partially skipping onto another ball caused the colliding balls to form an overlapping position, which looked unrealistic. Given more time, I would have liked to implement a system that detected potential nearby collisions, and decreased the speed at the final timestep if necessary to ensure that the balls collided exactly on the edges. Since the speed would only drop in the final timestep before collision, this change would be unnoticeable to the naked eye.

In general, the vector calculations were accurate, assuming balls collided only on edges and could handle minimal rounding down. However, the potential high speed of balls meant they did not collide on just edges, and it was during these overlaps that unrealistic collisions occurred.

4.1.3 Cue Collision Checker (Zareen Choudhury)

The cue collision checker was implemented very similarly to the ball collision checkers. The primary difference was that the cue collision checker was triggered only by the cue. Its inputs were the cue front's x- and y-coordinates, provided by the cue position tracking module, and the cue's x- and y-speeds, provided by the cue speed calculator. The squared distance between the colliding sprites was smaller as well, since the collision was occurring between a ball and the cue instead of two balls.

4.1.4 Pocket Checker (Matt Basile)

The pocket checker was a refactor of the ball-collision checker. Each pocket had a series of collision modules with each ball which continually checked for possible collisions. If a collision was detected, a flag signal was sent to the position handler, which then moved the pocketed ball off screen and halted its movement. This was a simple workaround we developed to keep other balls from interacting with a pocketed ball.

4.1.5 Position Handler (Matt Basile)

The position handler was the "brain" behind all of the game physics. It stored the ball positions, took in the new speeds and triggered inputs from all of the collision checkers (wall, ball, cue), and output the new ball speeds and positions for any given time step. Every ball was stored as an x and y coordinate for its center, and an x and y vector representing its direction and speed. By listening for every possible collision, the position handler selected which input to choose from when updating each ball's speed. If the position handler received a wall collision input for ball 1, it selects the updated speed from that wall collision checker and used that speed to update the position of ball 1 in the following timesteps. This was achieved through a series of conditional statements, one of every collision. Each conditional statement was also gated by additional conditional statements, which helped impose a hierarchy on collisions and other operations. At the highest level was the pocketed flag: if a ball had rolled into a pocket, it was removed from all future physics calculations to ensure it accidentally did not roll back on screen. Furthermore,

wall collisions were given priority over ball collisions, since an undetected wall collision would send a ball of screen, while an undetected ball collision simply sent a ball somewhere randomly on the court. We achieved this by including a lack of wall collisions as part of the conditional for checking for new speeds.

Our reasoning for storing the balls the way we did was mostly based around simplicity and ease of calculations. As mentioned Section 4.1.2, we wished to avoid the use of square roots and divisions (except division by powers of 2) operations throughout our code. Storing ball speed as a separate x and y vector kept us from having to use trigonometry to separate vectors, and instead use simple operations such as dot products and multiplications in our calculations.

The position handler evolved heavily throughout the course of the project. In the initial proposal, the position handler only updated the positions and speed of the balls. The actual positions of the balls were stored in the game FSM and sent to the position handler, which updated the ball positions according to the speed and then decreased the speeds (due to friction). However, we discovered that it was simpler and cleaner to simply store the ball positions in the same place they were being updated. Furthermore, when we split from single, unified collision checkers to individual modules, we needed a way of consolidating all the different possible inputs, since a single ball needed to listen to possible collisions with many different objects. Centralizing all these inputs in the position handler allowed us to organize our code and consolidate all position and speed related inputs to a single location. While the position handler did grow very large, much of the code was very similar, and thus easier to implement than some other modules. The greatest error we ran into was bug fixing due to the length of the code. In general, we minimized this by keeping the code as organized as possible and isolating which balls were performing incorrectly, and isolating not only where that ball's inputs could be missing, but also where they could have been added accidentally.

4.1.6 Friction (Zareen Choudhury)

The intent of the friction module was to reduce the speed of the balls and bring them to rest after they had been set into motion. Each ball was associated with a unique instance of the friction module. Speed reduction was implemented by decrementing a ball's velocity geometrically. When a ball started moving, its initial x- and y-velocities were stored. Those velocities were decremented by 1 pixel/frame every N clock cycles. The number of clock cycles, N, was inversely proportional to the velocity; a larger velocity would be decremented faster, and vice versa. The x- and y-components were decremented independently. If the ball collided with another ball or wall, the new post-collision velocities replaced the initial velocities and used to recalculate N. When both components were reduced to a magnitude of 0, the ball was put to rest and set as inactive.

Based on testing through visual inspection, I found that this implementation of friction did not result in very realistic or smooth speed reduction. Since the x- and y-components were reduced separately, oftentimes one component would be reduced to zero before the other. This resulted

in the last several clock cycles of the ball's movement being in only the horizontal or vertical direction, which did not appear very realistic.

A more physically accurate model for friction is to reduce both x- and y-components by the same multiplicative constant, as opposed to an additive constant. Unfortunately, we began implementing this model with too little time to spare and were unable to successfully integrate it with the rest of our code. If we had more time, we would want to integrate this implementation to have a smoother and more accurate friction model.

4.2 Game FSM (Zareen Choudhury)

Figure 4.2 illustrates the configuration of the pool table. There were two players in the game - stripes and solid. The pool pieces consisted of one white cue ball, two striped balls (light blue and dark blue), and two solid balls (red and orange). Only the cue ball could be hit by the cue stick. There were six pockets, corresponding to the four corners, the center top, and the center bottom. The pool table also contained a small circular sprite (black in Figure 4.2) indicating the virtual position of the front of the cue stick. The LED hex display on the labkit was used to indicate the stripes and solid teams' current scores.

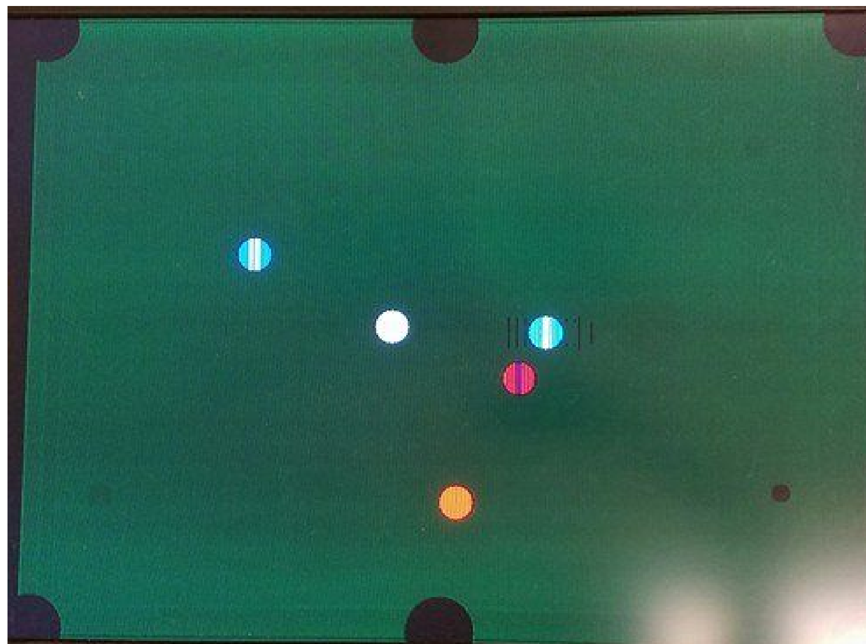


Figure 4.2: Initial pool table

The pool game FSM is depicted in Figure 4.3. In the Start Game state, all the balls were initialized to their starting positions with starting velocities of 0 (as shown in Figure 4.2). After one clock cycle, the FSM moved from Start Game to Track Cue Stripes. The player turn was initialized to the stripes team. The color of the cue sprite was blue in this state to indicate the stripes team's turn, since both striped balls were blue.

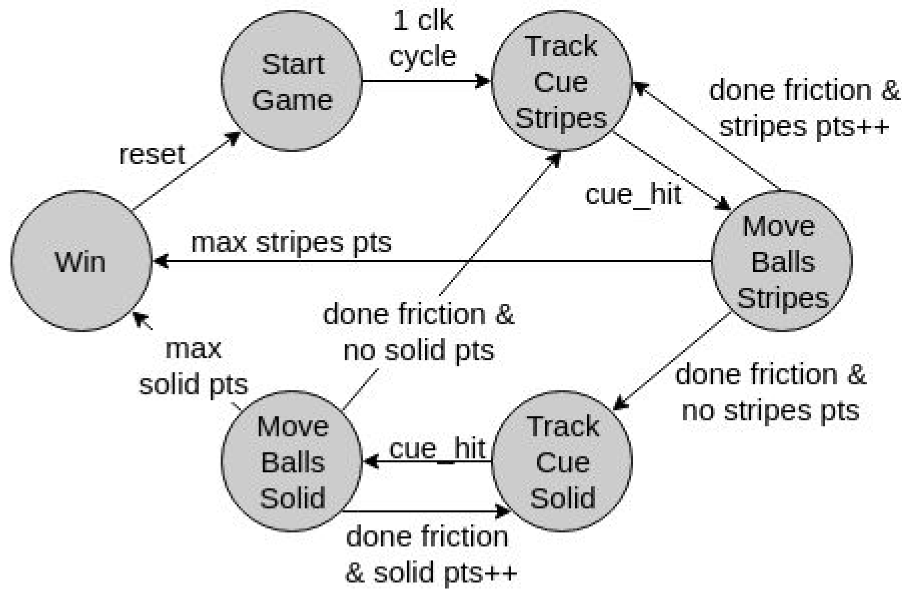


Figure 4.3: Pool game FSM.

When a player hit the cue ball with the cue stick, as detected by the cue collision checker, the FSM transitioned to the Move Balls Stripes state. In this state, the cue was no longer tracked for cue collisions, as the balls moved and collided among themselves until they all reached a rest state. Accordingly, the cue sprite's color was changed to black to indicate that it was not active in this state. Once all the balls stopped moving, as determined by the friction modules, a signal was sent to the FSM.

At this point, the FSM compared the current number of points with the number of points before the round. If the stripes team scored a point by pocketing a striped ball, the FSM would loop back to the Track Cue Stripes. Otherwise, if the stripes team did not score any points, pocketed a solid ball, or scratched (pocketed the white cue ball), it would be the solid team's turn, and the FSM would instead transition to Track Cue Solid.

If it was the solid team's turn, the cue sprite turned red to indicate their turn. An analogous set of transitions occurred in the Track Cue Solid and Move Balls Solid states as in Track Cue Stripes and Move Balls Stripes. At the end of both the Move Balls Stripes and Move Balls Solid, the FSM also checked whether the given team had pocketed all their balls. If they had, it would transition to the Win state, which displayed which player won (Figure 4.4). The game could be reset to Start Game by pressing the enter button.

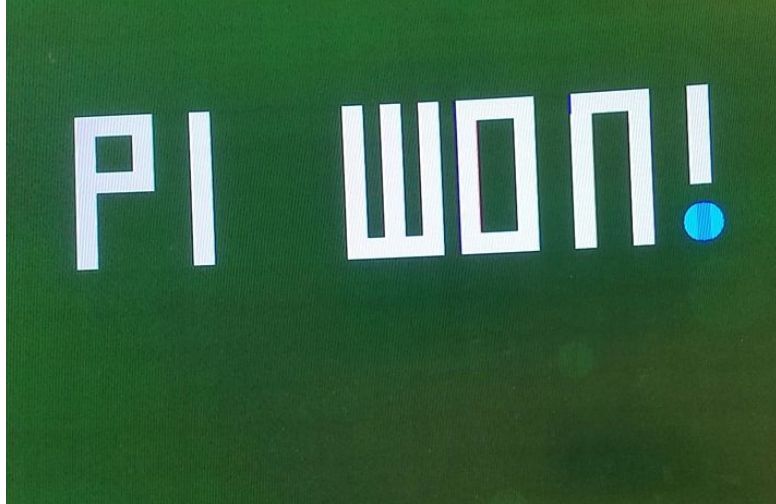


Figure 4.4: Pool game win screen.

4.3 Game Visuals (Matt Basile)

All visual components of the game used the same "sprite" code, similar to what was used in Lab 3. Depending on the current value of hcount and vcount, the sprite module either outputted that sprite's color, or black. These pixel outputs were then summed, and if their sum equaled 0, that pixel was set to be the background color of green. This ensured that we would have an even background color that would not interfere with any of the ball or pocket colors.

All round components were stored by their center x and y positions and their radii. A pixel was said to be in the circle if its distance from the center x and y was less than the radius. Since the radius of each ball was fixed, we could avoid having to use a square root by calculating $(x\text{-}x_b)^2 + (y\text{-}y_b)^2 = radius^2$. Input variables let us designate a ball's color, pattern, and player. The cue stick marker used a slightly modified version of this ball sprite code with a different radius parameter and a changing color input, to match each player's turn.

The final win screen simply used a series of rectangular sprites to simulate letters. If given more time, I would implement a more complicated text display system. This would allow us to display messages such as "Press Enter to Play Again", or each player's turn and score on the pool table during the game.

5 Lessons Learned

5.1 Matt Basile

I believe one of the biggest lessons I learned was the importance of structuring and organizing code, especially in Verilog. A fair portion of this project's difficulty came from keeping track of all

of the different inputs, outputs, and modules. Because almost every module we wrote in some way interacted with every other module, keeping track of everything was certainly a challenge. Thus, even small things such as comments, indenting, and module layout were very important to ensuring we could debug in a timely manner.

Furthermore, I recognized that working with Verilog, and hardware in general, presents a unique set of problems that I am unaccustomed to working with in computer science. Specifically, timing is extremely important in Verilog, and having to work with discrete cycles and blocking assignments while coordinating inputs from other modules is always a challenge. I learned to keep things simple whenever possible, and use state machines for everything else. This is also why moved the actual calculations of the collisions to individual models - this allowed us to parallelize the process even further and let the position handler run in a single clock cycle, as opposed to multiple clock cycles for each calculation.

The importance of having a detailed, yet flexible, starting plan was also very apparent throughout this lab. Our final block diagram looks quite different than our initial one, but just the presence of that first diagram played a key role in keeping us organized in the early days, and also when we moved around variables and processes between modules. Seeing where everything connected helped us know how to reconnect inputs and outputs, and ensure that information flowed through our new chart.

Observing the sheer variety of projects the other students completed, alongside the complexity of our own, has also reaffirmed my appreciation for the FPGA has an incredibly versatile tool.

5.2 Zareen Choudhury

One piece of advice I have is to start integrating code earlier. While this may not necessarily apply to every project, in our case we would have benefitted from integrating the external and internal components before we did. I would also encourage future students to set up and use version control (e.g. Git) from the beginning. We did not have Git set up properly, so we ended up saving and sending each other various versions of our code. Many other groups used a similar homegrown Git approach, but it would have been significantly easier to use some form of version control instead.

I wish I had made my code more ready for change from the beginning, as well. This includes using parameters for hard-coded numbers so that they can be adjusted easily and structuring code to be modular so that additional instances can be added easily. In order to add more balls, we had to refactor our code to make it more modular. This process made us lose several days of work, since there were many bugs to fix in the refactored code. If we had made the code modular to begin with, we could have saved this time.

Finally, I advise students to talk with other students and seek out feedback from the staff. It ends up that many projects have similar elements. In my case, another group was working with

an accelerometer, and another one was using IR tracking. It was very beneficial for me to talk with them through the design process and to help each other debug. I learned a lot from working in this collaborative manner.

6 Future Work & Conclusion

Overall, we were successful in creating a pool game that integrated real-world interaction using a cue stick with a virtual pool table. We implemented accurate and real-time cue tracking and speed calculation, as well as real-time ball position updates. We originally predicted that achieving a fast response time would be our biggest challenge, but were able to overcome this by eliminating the need for heavy math calculations and adjusting the clock. The ball position updates were mostly accurate at low- to mid-level velocities. The game included most elements of a typical pool game, including alternating team turns, pockets, scratches, and scores.

If we had more time, we would want to clean up the ball position updates and implement some stretch goals we had. As mentioned earlier, the friction model could be made more realistic by reducing speed with a multiplicative instead of an additive constant. Many of the ball and wall collisions appeared did not occur as expected at higher velocities, which we would want to clean up. We could expand the game to include the full number of pool balls, which would be easy to scale with our modular organization. Additional features could include displaying score with sprites on the pool table, instead of the hex LED display, and adding collision sounds. These additions would make the game experience even more seamless and advance the goal of blending the line between physical and virtual entertainment.

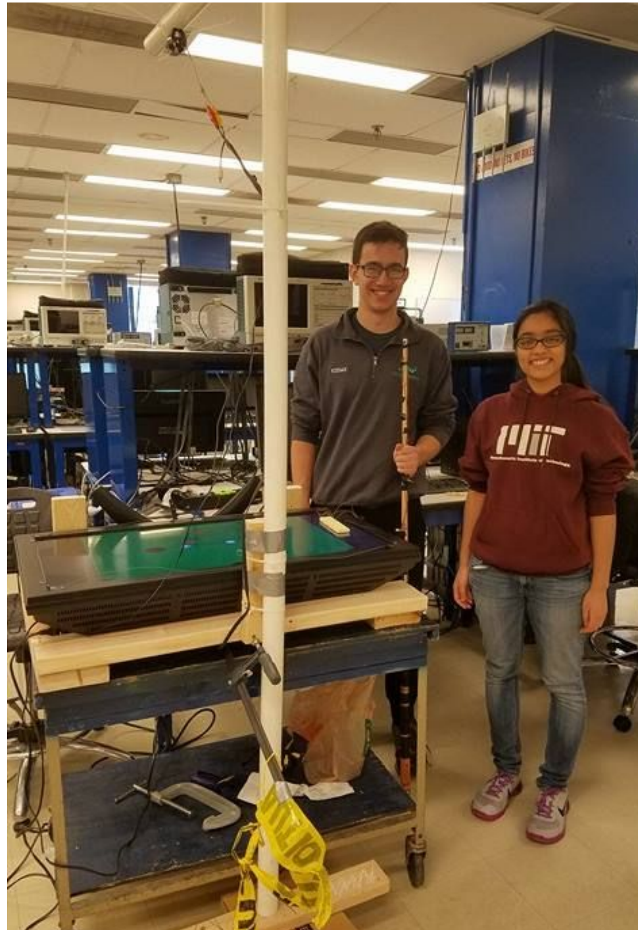
7 Acknowledgements

We would like to sincerely thank the 6.111 course staff for supporting our project. Prof. Gim Hom was instrumental in providing materials like the TV, NTSC camera, IR filter, and countless LED's. He offered key insights for working with the NTSC camera for tracking, adjusting the resolution of the output for calibration, and designing the overall hardware setup. His enthusiasm for our project was infectious, and motivated us to push this project as far as we could take it.

Our project mentor, Joe Steinmeyer, provided the MPU-9250 along with other electrical components that were essential to our project. Joe's experience with external components was extremely helpful for setting up and testing the accelerometer. He offered insightful suggestions for implementing friction and pool ball physics without costly mathematical operations. As a mentor, Joe frequently checked in on our project and and provided useful feedback.

Finally, we would like to thank the remaining TA's and LA's: Weston Braun, Mitchell Gu, Shawn Jain, Alex Sloboda, David Gomez, Valerie Sarge, and Madeline Waller. Thank you to each and every member of the 6.111 course staff for your suggestions and advice, for your debugging

help, for keeping lab open after hours and over Thanksgiving break, and for your encouragement, enthusiasm, and humor. La PC-na wouldn't have been possible without you!



Thank you 6.111!