6.2050 Project Final Report – ByteBall

Srinidhi Venkatesh
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
svenkat@mit.edu

Justin Malloy
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
jamalloy@mit.edu

Riley Contee
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
ricontee@mit.edu

I. Introduction - JUSTIN

We are designing a physics engine with a pool table application. Our general goals include: 1) Modeling billiard ball motions and interactions with each other and their surroundings, and 2) Utilizing image detection to take user input as a force acting on a cue ball. To begin the project, we created a Python simulation to validate our physics modeling of collisions and to picture how the balls' sprites will need to change based on their rotational movement. The challenges in translating this design to hardware lie in simulating the accuracy of collision modeling, doing real-time data processing, synchronization, and resource management.

Our system models the translational and rotational motion of billiard balls, collisions with momentum transfers, and frictional effects on the balls' trajectories as they move across the table and interact with walls. We want the final product to include user interaction from a physical cue stick and image detection with a camera. Our approach incorporates modules to highlight collision detection, translational and rotational dynamics, and position tracking. We are using BRAMs to store image sprite information about our interacting billiard balls and for a frame buffer. We evaluate the performance of our physics engine via visual appearance and behavior of the simulation and efficiency through BRAM and FPGA resource usage.

II. Project Requirements - SRINIDHI

We will meet our project requirements by creating, testing, and integrating modules dedicated to modeling the physical components and interactions between balls.

A. Commitments

- Modeling translational motion in x and y dimension
- Modeling rotational motion about z axis
- Modeling translational collisions between multiple balls
- Modeling rotational collisions between multiple balls
- Modeling translational and rotational collisions with walls

B. Goals

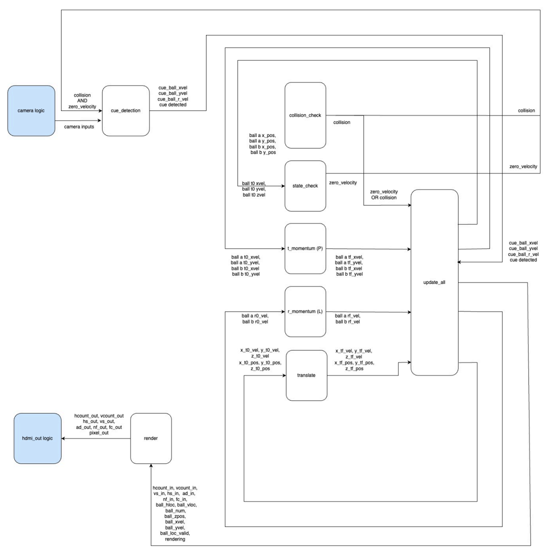
- Cue detection with camera above the monitor and image detection of the cue stick
- Translating cue stick motion to cue ball motion
- Model striped balls on pool table, including mapping rotations (ultimately, ended as a stretch goal)

C. Stretch Goals

- 3D rolling animation for the balls.
- Nice to have: Add pockets and disappearing billiard balls.

III. High Level Design Overview - RILEY

Figure 1 outlines our block diagram which represents the modular design of our physics engine application, and Figure 2 outlines our FSM design to control the interactions between the billiard balls and the pool table. The FSM closely resembles the block diagram as it shows the interactions between the modules we use to manage the physics calculations of the billiard balls. The FSM begins with cue detection – once a force is detected from our user, upon a successful collision, it will be applied to our cue ball in the system, kicking off the physics engine. It is important to note that the modules are applied to all pairs of balls for collision check, colliding pairs for momentum calculations, and individual balls for state check, translation, and render.



camera logic
collision AND zero_velocity
camera inputs
cue_detection
cue_ball_xvel cue_ball_yvel cue_ball_r_vel cue detected
collision_check
collision
ball a x_pos, ball a y_pos, ball b x_pos, ball b y_pos
zero_velocity
state_check
ball t0 xvel, ball t0 yvel, ball t0 zvel
zero_velocity OR collision
t_momentum (P)
ball a t0_xvel, ball a t0_yvel, ball b t0_xvel ball b t0_yvel
ball a tf_xvel, ball a tf_yvel, ball b tf_xvel ball b tf_yvel
update_all
r_momentum (L)
ball a r0_vel, ball b r0_vel
ball a rf_vel, ball b rf_vel
translate
x_t0_vel, y_t0_vel, z_t0_vel x_t0_pos, y_t0_pos, z_t0_pos
x_tf_vel, y_tf_vel, z_tf_vel x_tf_pos, y_tf_pos, z_tf_pos
hdmi_out logic
hcount_out, vcount_out hs_out, vs_out, ad_out, nf_out, fc_out pixel_out
render
hcount_in, vcount_in, vs_in, hs_in, ad_in, nf_in, fc_in, ball_hloc, ball_vloc, ball_num, ball_zpos, ball_xvel, ball_yvel, ball_loc_valid, rendering

Fig. 1. Block Diagram for ByteBall

IV. Modules - ALL CONTRIBUTED

The following is an overview of the modules in our system.

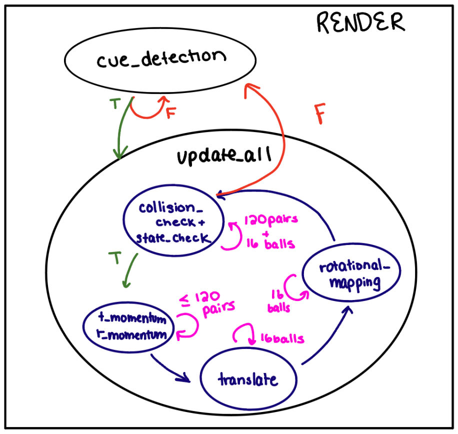


Fig. 2. General FSM Design for ByteBall

A. `update_all_fsm` - *SRINIDHI*

The `update_all` state machine has 6 main states: `IDLE`, `ENTER_FSM`, `STATE_CHECK`, `COLL_CHECK`, `MOMENTUM`, `TRANSLATE`. Most of these states directly map to their namesake module. However, the `IDLE` state manages when the system enters the FSM. It will move onto the `ENTER_FSM` state, if and only if a cue hit is detected while transfering the cue's x, y, and z velocities to the cue ball. Once in the `ENTER_FSM` state, we check for a new frame and enter our FSM, beginning with `STATE_CHECK`. In `STATE_CHECK`, we check the x, y, and z velocities of each ball to determine if they have become stationary. If so, we set our state check signal high, else it is low. Following this stage, we move to `COLL_CHECK`. In the `COLL_CHECK` state, we iterate through each pair of balls and check for pairs that are actively colliding in the current frame. We store those collisions using one hot encoding in a register to be used in the following `MOMENTUM` stage. If no collisions are found, we set our more collisions signal low. That said, if the collision signal is low and the state check signal is high, we are ready for a new turn and circle back to the `IDLE` state. Else, we transition to `MOMENTUM`. In `MOMENTUM`, we run both the `t_momentum` and `r_momentum` modules. In these modules, we update the x, y, and z velocities of balls that are colliding with each other. If a collision was detected in the previous stage, we compute the new velocities according to `t_momentum` and `r_momentum` descriptions in following sections. Once we have calculated and stored the updated velocities for the colliding pairs, we transition to the `TRANSLATE` stage. The `TRANSLATE` module handles the translational/rotational velocities and to account for wall collisions and friction, as well as outputs the updated the x, y, and z positions to render. `TRANSLATE` runs for each individual ball on the screen. Once finished, we transition back to `STATE_CHECK` to continue cycling through the FSM, or exit and enter the `IDLE` state to wait for the next cue input.

In terms of code organization, this FSM calls the needed modules. Then, a combinational case statement assigns the modules' input variables to the appropriate input values using a ball count that is incremented within each module. Additionally, a sequential block assigns initial positions to the balls on the game board and sets initial ball counts to loop through the various combinations of objects. In this case statement, state transitions are set up, valid signals are set, and ball counters are incremented.

B. `cue_detection` - *SRINIDHI*

The `cue_detection` module takes camera input and utilizes our `center_of_mass` module from previous labs to detect the position of our cue stick at two different instances in time as the user pushes the cue stick across the pool table. The second instance is when the user's cue stick crosses the position and area on the screen that are attributed to the cue ball. This position change is then divided by the time interval for this motion to give us a horizontal and vertical velocity that will be transferred to the cue ball. Additionally, this module provides us the necessary data to draw a mask and cross hair on our detected cue stick point. This additional functionality allows for better user input to the system. An important functionality here is the implementation of a Xilinx BRAM to perform a clock domain crossing between our `clk_camera` and `clk_pixel`. This clock crossing is necessary as we need to scale our camera input positions to our pool table screen positions such that our user input is accurately translated between the two different frames. This scaling logic requires a preset calibration of the pool table pixel width and length in terms of the camera's pixels. This scaled ball position is then compared to our cue stick input positions to decide if `hit_detected` goes high or low. All of the position and velocity data for the cue ball is calculated and then transferred to our state machine when a hit is detected. The The `cue_detection` module is pictured in Fig. 3.

This module functions and is able to transfer an input velocity to the cue ball from the cue stick's motion. However, the current implementation of the module struggles to take a new input velocity using the cue stick movement after the game state has come to rest and is ready for the next cue detection check. The module instead moves the cue ball when a hit is detected, but with the same velocity as the previous hit. We have yet to resolve this integration bug for full game board functionality.

C. `collision_check` - *JUSTIN*

The `collision_check` module takes in the positions of a pair of balls and outputs whether they are colliding. This module requires a check of the distance between the two objects' x and y positions and ensures that they are separated by a distance of the sum of both balls' radii. This check is fully combinational and outputs a one bit high-low signal that signifies whether there is a collision or not. This module allows for confirmation of whether there needs to be a momentum

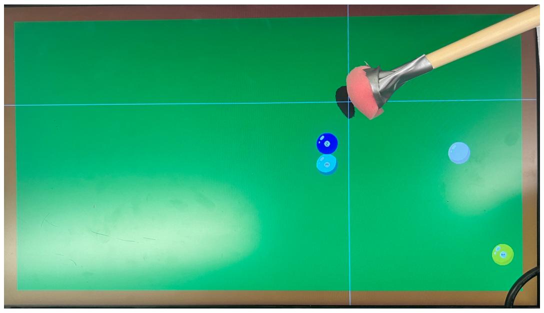


Fig. 3. *Cue Detection:* Cue detection mask (black blob below the cue stick point) and crosshair showing mapping of cue stick to pool table

calculation for a pair of balls. If there is no collision, then no momentum needs to be calculated for that specific pair of balls.

D. `state_check` - *JUSTIN*

The `state_check` module takes in the x, y, and z dimensional velocities of a ball and checks if each of those velocities is 0. If the ball is completely at rest, it outputs a one bit high-low signal that specifies whether the ball has zero velocity. This module is also fully combinational and allows for confirmation that every object on the table is at rest, allowing the state machine to either continue checking for collisions or transition to the idle state and wait for an input stimulus from the cue stick.

E. `t_momentum` - *RILEY*

The `t_momentum` module is responsible for updating the x and y velocities of a pair of colliding balls under the assumption of perfectly elastic collisions.

In our initial implementation the module took the following form: the module took in the x and y velocities and positions of two colliding balls, and outputted their updated x and y velocities as a consequence of their collision. The updated velocities were initially calculated using the following steps with sequential logic and extensive pipelining to account for the use of `sqrt` and `divider` modules.

- Compute the distance between the x and y positions of the colliding balls by subtracting the difference between input positions, dx and dy, respectively.
- Calculate the magnitude of the distance between the balls using an integer square root module [1], $magnitude = \sqrt{dx^2 + dy^2}$. NOTE: The maximum sum of the distances squared can be $(2 * BALL\_RADIUS)^2$.
- Use our divider module from previous labs to find the normalized vector for the x and y directions and extract the normal components for the x and y velocities.
- Finally, calculate the updated velocities where if one ball is stationary, the velocity is evenly split in the same direction among the balls, else send each ball in the opposite direction with other's velocity.

During integration, we found some bugs and difficulty in using the positions of the colliding balls to update their departing velocities as we did in our Python simulation. We instead focused on creating the "bounce away" effect and ignored the vector normalization using the balls' positions. However, we used the same behavior to transfer and split velocities to maintain the elastic collisions where $v_{1,i} + v_{2,i} = v_{1,f} + v_{2,f}$, since our balls have equal mass. These changes allowed for the module to become fully combinational.

F. `r_momentum` - *RILEY*

The `r_momentum` module is responsible for updating the z rotational velocities (spin) for a pair of colliding balls under the assumption of a perfectly elastic collision. The module takes in the current z velocities of the two colliding balls, and outputs their updated z velocities as a consequence of their collision. The updated velocities are calculated by considering 3 cases: 1) `BALL_A` is spinning and `BALL_B` is not, 2) both balls are spinning, and 3) both balls are not spinning. In Case 1, the z velocity from the spinning ball will be split evenly among the two colliding balls. In Case 2, even if the balls are spinning in the same or opposite direction, we want to average their z velocities and split among the two balls. In Case 3, after the collision, both balls will continue with no spin.

G. `translate` - *RILEY*

The `translate` module takes in the ball's current x, y and z positions and velocities, and outputs its updated x, y and z positions and velocities as affected by wall collisions and friction. The module is responsible for checking boundary conditions using the calculated next position of the ball (taken from the current position and velocity). The module will then ensure visually, that the ball remains within the boundary and also flips the direction of the velocity to bounce off the walls. Additionally, the module factors in translational friction as the balls move across the table (acting on the x and y velocities) and collide with walls (acting as rotational dampening on the z velocity). Given the updates to the x, y and z velocities, we can compute the next x, y and z positions to output.

H. `rotational_mapping` - *RILEY*

The `rotational_mapping` is responsible for handling sprite rotation mapping, so as we render sprites it can visually appear to be rolling. The module takes in the x, y position of a pixel in the sprite and the current z rotational velocity and outputs the updated x, y position and a valid signal. The module will include a variation of the CORDIC Algorithm [2] (currently under *trig_funcs* in the repo) to produce the vector rotation for the sprites.

It is important to note that in our final design we did not have the chance to fully integrate `rotational_mapping` into the FSM, and instead used multiple rotated sprites and swapped between them based on the z position of the ball.

I. `render` - *JUSTIN*

The `render` module manages the pipelining of video signals such as `h_count`, `v_count`, `h_sync`, `v_sync`, and pixel values to be output via HDMI. This module also has the responsibility of maintaining a copy of the position of each ball in order to display the sprite in the appropriate position on the screen. The module will also keep track of how each ball should be rotated to properly animate the spinning effect when displayed.

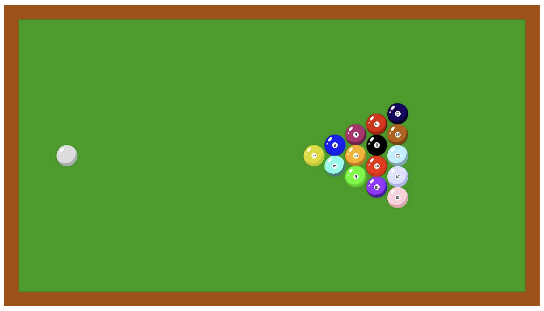


Fig. 4. Render Module Output on a Monitor

A sub-module of render, `display_ball`, computes the updated location of the number sprite on a ball based on the current x and y location of the sprite and the updated position using the ball's velocity. If the sprite reaches the edge of the ball, we signal that it has crossed the edge and needs to be rendered at a new location from the opposite edge of the ball. Given that the position is updated using the ball's velocity, the rotational motion is based on the x and y directions the ball moves in. Additionally, we want to note that each number to render is stored in BRAM, with a total of 8 physical BRAMS (we store two sprites per BRAM by taking advantage of the two read ports) dedicated to the balls. Figure 4 shows the render of a typical 16 ball set up of pool.

J. `top_level` *- SRINIDHI*

The top level takes in camera inputs and I2C lines, as well as a 100mHz clock and physical inputs. The module outputs HDMI signals for the monitor. In this module, we first call the cue detection module with the various camera modules fed in. We also create our camera and pixel clocks that we feed into cue detection to transfer information between the camera on the camera clock and our monitor on the HDMI clock. We create an instance of `pixel_reconstruct` in this module before feeding its outputs to `cue_detection`. We then feed the output velocity from this module into our `update_all` FSM. Additionally, we create our rendering and video signals and call our `video_sig_gen` module and `render` module to render our balls to the monitor. Additional signals include checking for our crosshair pixels and setting the mask color for that accordingly. This module also contains all our camera registers and TMDS modules. Overall, this module houses camera signals, `cue_detection`, and the `update_all` FSM to interface with user inputs.

V. Design Evaluation - ALL CONTRIBUTED

Due to the physics calculations being done during the long frame blanking region, timing and throughput is not something to be concerned about for the design. We take advantage of this time to reuse many resources such as the `translate` and `momentum` modules to be efficient on the number of LUTs in use and the amount of data being transmitted at any one time.

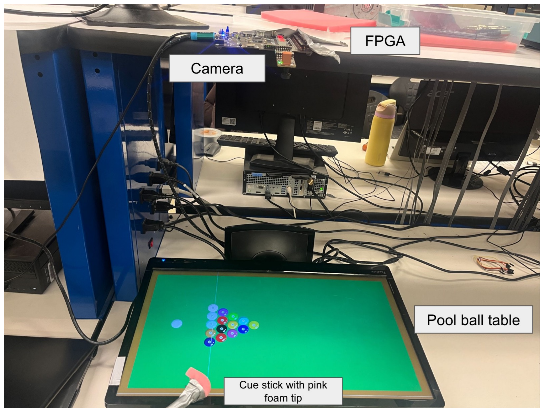
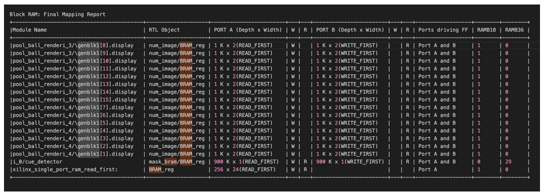


Fig. 5. General Set Up for ByteBall

Unfortunately, this design philosophy fails in one place, and that is, with render. With rendering needing a high throughput, we must use a BRAM for each ball, making the total number of BRAMs in use 16 for the whole design. All BRAMs use a single read port so Vivado will optimize pairs of BRAMs into one physical BRAM, leaving us using 8 BRAMs for rendering. This problem is also apparent for DSP usage as calculating the address for the sprite BRAMs requires a DSP which means that we use 16 DSPs for the rendering module.

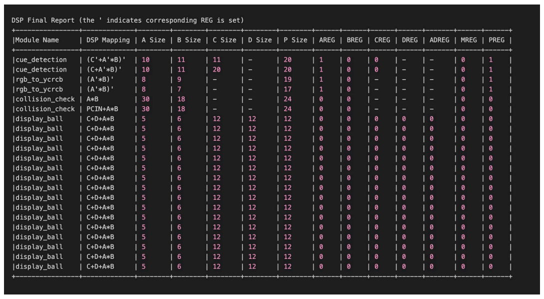
Our previous design which only contained `translate` and `render` did not use any DSPs outside of the render module. At that moment, we anticipated the other modules would need to use DSPs such as `translate` and `rotational_mapping`. Also, when those modules would integrated into the design, we would see the benefit of our design based on the number of DSPs used.

According to the Synthesis Report of our build, we utilized a total of 22 DSPs for the modules associated with cue detection, collision check, and render. Additionally, we utilized 8 BRAMs for ball rendering, 1 BRAM to account for clock crossing between our `clk_camera` and `clk_pixel`, and another 29 BRAMs for cue detection frame buffer.


```
Block RAM: Final Mapping Report
+-------------------------------------------+--------------------+---------------------------+---+---+----------------------------+---+---+------------------+--------+--------+
|Module Name                                | RTL Object         | PORT A (Depth x Width)    | W | R | PORT B (Depth x Width)     | W | R | Ports driving FF | RAMB18 | RAMB36 |
+-------------------------------------------+--------------------+---------------------------+---+---+----------------------------+---+---+------------------+--------+--------+
|pool_ball_renderi_3/\genblk1[8].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[9].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[10].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[11].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[12].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[13].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[14].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_3/\genblk1[15].display   | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[7].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[6].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[5].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[4].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[3].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[2].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|pool_ball_renderi_4/\genblk1[1].display    | num_image/BRAM_reg | 1 K x 2(READ_FIRST)       | W |   | 1 K x 2(WRITE_FIRST)       |   | R | Port A and B     | 1      | 0      |
|i_0/cue_detector                           | mask_bram/BRAM_reg | 900 K x 1(READ_FIRST)     | W | R | 900 K x 1(WRITE_FIRST)     |   | R | Port A and B     | 0      | 29     |
|xilinx_single_port_ram_read_first:         | BRAM_reg           | 256 x 24(READ_FIRST)      | W | R |                            |   |   | Port A           | 1      | 0      |
+-------------------------------------------+--------------------+---------------------------+---+---+----------------------------+---+---+------------------+--------+--------+
```
Fig. 6. Utilized BRAM

VI. Implementation Insights - ALL CONTRIBUTED

We began the design of our physics engine with a Python simulation to understand the foundations of the physical interactions between balls. This approach allowed us to focus on



```
DSP Final Report (the ' indicates corresponding REG is set)
+----------------+--------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|Module Name     | DSP Mapping  | A Size | B Size | C Size | D Size | P Size | AREG | BREG | CREG | DREG | ADREG | MREG | PREG |
+----------------+--------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|cue_detection   | (C'+A'*B)'   | 10     | 11     | 11     | -      | 20     | 1    | 0    | 0    | -    | -     | 0    | 1    |
|cue_detection   | (C+A'*B)'    | 10     | 11     | 20     | -      | 20     | 1    | 0    | 0    | -    | -     | 0    | 1    |
|rgb_to_ycrcb    | (A'*B)'      | 8      | 9      | -      | -      | 19     | 1    | 0    | -    | -    | -     | 0    | 1    |
|rgb_to_ycrcb    | (A'*B)'      | 8      | 7      | -      | -      | 17     | 1    | 0    | -    | -    | -     | 0    | 1    |
|collision_check | A*B          | 30     | 18     | -      | -      | 24     | 0    | 0    | -    | -    | -     | 0    | 0    |
|collision_check | PCIN+A*B     | 30     | 18     | -      | -      | 24     | 0    | 0    | -    | -    | -     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
|display_ball    | C+D+A*B      | 5      | 6      | 12     | 12     | 12     | 0    | 0    | 0    | 0    | 0     | 0    | 0    |
+----------------+--------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
```

Fig. 7. Utilized DSPs

the physics of the collisions, momentum transfers, and how to visualize the moving sprites with the balls' movements. In the Python simulation, we were able to refine our collision detection and momentum transfer calculations to be implemented on the hardware. This step was beneficial in having a baseline to verify the correctness of our physics model without initially dealing with hardware integration.

When we transitioned to the FPGA implementation, the foundational physics calculations remained the same for the translational and rotational motion and momentum transfers. However, moving those algorithms into hardware introduced other challenges and implementation insights.

One of the most noticeable insights was the need for synchronization and pipelining, especially in the translational momentum module. To calculate the updated velocities for a pair of balls during a collision, we needed several intermediate calculations. In the module, we used square root and divide operations to compute the normal vector, so we needed to use integer square root and divider modules. Using those required us to complete extensive testing to pipeline the module to ensure minimal delay between calculations and accuracy.

We also worked to integrate translational momentum which proved challenging since it was one of our few sequential modules that also required multiple dividers and square root modules. We wrote this module based on vector math and attempted to pipeline it. We also incorporated a version of shifted fixed point math to get normal vectors. However, as we robustly test benched this module, it failed certain edge cases. In attempting to fix this module, we changed various variable lengths, performed simplifications to our calculations, and rewrote our square root module from scratch. However, we eventually switched over to an estimation of momentum behavior rather than using exact vector math. This new logic we implemented and integrated is the same as the core of our Python simulation's logic. If one ball is stationary, we would like to simply split the moving balls velocity between the two in opposite directions. Else, when both balls are moving towards each other, we want to send them in opposite directions with the other's velocity.

Another challenging implementation process was from our integration of cue detection into our system. We had to modify our camera code from previous labs to incorporate it into our existing module structure. We set up a BRAM for storing our mask pixel data and pipelined all of our addresses and valid signals. We initially planned to use a FIFO to pass through our mask signals for rendering but eventually switched to the xilinx BRAM for simplicity. The overall application of this module was challenging as it involved many inputs being moved around while we were also performing output data calculations. We also realized that we would need to scale our camera pixel based center of mass calculation to correspond with the right position on our screen's pixels. This scaling logic required additional calibration and fine tuning to find an optimal position for our monitor set up. As we implemented this interaction, we also realized we had to handle clock domain crossings. This module ended up being much more involved in its creation than we originally planned, which led to a more involved debugging and integration process.

The integration process required slow additions of each module into our top-level code with much debugging required for edge cases and unforeseen interactions. We discovered many bugs throughout this integration process and had to revisit many of our modules to edit or even rewrite them to better match our needs. The debugging process also became much lengthier during integration as it required many builds to be able to see how different modules interacted. Overall, we met most of our module implementation goals and many of our integration goals.

If we were to do such a project again, we would shift our design process away from writing all the modules first and then integrating to integrating each module as we write them. We ended up needing to rewrite or majorly edit many of our original modules again during our integration process anyways. Knowing what integration looks like as we write the modules would allow us to be more efficient in our design flow.

Additionally, it would have been beneficial to get a jump start on the rotational mapping logic to fully implement and integrate it properly into our system. To implement the CORDIC algorithm for our system it needed a few alterations to account for the centering offset of the image sprite. Next time, early on we could spend time debugging the normalization for the offset, adjusting the quadrant logic, and assessing the edge cases.

VII. Code Repository - Justin

The following repository contains Verilog for the design in progress, as well as the Python code for testing the hardware, and for simulating the physics.

[redacted]

VIII. Contributions - ALL CONTRIBUTED

First, and foremost, we would like to acknowledge Stephen Kandeh, our mentor, for his guidance throughout the project when refining our design approach. We'd also like to express our gratitude to the entirety of the 6.2050 course staff for the support and feedback provided in office hours to help our project come to life. We would like to also acknowledge the academic references and documentation that helped inform our

design approach and implementation of foundational modules such as divider, square root, and CORDIC algorithm, as cited.

This project was a collaborative effort by Justin Malloy, Srinidhi Venkatesh, and Riley Contee. All three of us played a critical role in the design, implementation, and evaluation stages of creating our FPGA-based physics engine. We divided the work among the essential modules needed to complete the engine. In the design stage, when creating the Python simulation, Justin focused on the updating of the balls' sprites based on the movement of the ball, Srinidhi focused on establishing the physics needed to implement simplified momentum calculations, and Riley focused on implementing the rotational and translation momentum transfers between colliding balls. When we transitioned to hardware, Justin focused heavily on creating and testing the complex render module (sprite generation, sprite overlay, etc.), module integration, and on the image detection aspect of the project. Justin played a large role in integration and testing of the physics engine. Srinidhi focused on implementing and testing the critical translation module for wall collisions, friction modeling, translational position updates, as well as determining data structures and methodologies for intermediate storage of values within one play of the game. Riley focused on the collision check, state check, and rotational momentum module. Riley and Srinidhi also collaborated on writing and testing the translational momentum module which had several versions across debugging attempts. Srinidhi also mainly worked on setting up and testing camera usage for cue detection, center of mass calculations, and camera to screen scaling. She also worked on integration of cue detection into the overall engine at large. As this was a team effort, we all shared the responsibility of debugging modules and helping out where needed.

References

[1] "Square Root in Verilog." Project F - FPGA Development, 22 Dec. 2020, projectf.io/posts/square-root-in-verilog/.

[2] "Verilog code for sine cos and arctan using CORDIC Algorithm." VLSI Universe , 3 June 2021, https://www.vlsiuniverse.com/verilog-code-for-sine-cos-and-tan-cordic/.