Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.111 Final Project Report

2-Player Virtual Air Hockey

Alex Chen, Abraham Shin, Yuqing Zhang

December 12th, 2014

Contents

---

1 Introduction

We built a 2-player virtual air hockey game. Our goal was to implement an exciting and easy-to-play game without the burdensome physical setup. The players are able to control the mallets by moving two colored squares on a desk. On the screen, we render two perspectives of our virtual air hockey table, one for each player.

We picked this project for two reasons. Firstly, implementing air hockey is fun. Air hockey is a timeless classic that we grew up with, and reliving our childhood was one reason our project was fun to work on. Building a game also introduced an interactive component of our product that other people could enjoy. Second, air hockey presented interesting technical challenges. For example, the implementation of a realistic physics engine and the generation of smooth gameplay requires efficient design and attention to detail. This game also allowed for flexible reach goals, since there is always room for additional features.

Air hockey works as follows. In physical air hockey, two players control mallets that are used to strike a puck, which slides across a rectangular table with minimal friction. The puck bounces off the four table sides, which are raised to form walls. Along the two shorter sides of the table are two goals, one defended by each player. The objective of the game is to hit the puck into the opponent's goal. When a goal is scored, the scoring player wins a point.

In retrospect, we found this project challenging to implement cleanly. Smooth gameplay requires handling noisy camera inputs well. The physics was nontrivial to simulate. For example, the speed of the puck varies considerably throughout a game in an actual air hockey game. Given that we have a table size restriction in pixels and collision detection has to be done in digital circuits, the movement of pucks were hard to control at high velocities. Further, we were not able to reach our original goal of implementing 3-dimensional graphics. After making some progress, we ran into some unexpected problems we were not able to overcome and settled on 2-dimensional graphics instead.

Overall, however, we were still able to produce a working prototype (in time for the video recording, but unfortunately not in time for the checkoff). The object recognition module was successful in providing a smooth movement for users' mallets while the physics module could speed up or slow down the puck using the mallets' momentum. Our clean graphics interface produced nostalgic retro-style fonts and color schemes.

2 Summary

We designed our system so that members of our team could effectively work in parallel, such that individuals would be minimally bottlenecked by the other members until the final integration. We separated our system into four abstract components that interfaced with one another: object recognition, physics engine, graphical interface, and game logic (Figure 1).

The object recognition module is responsible for interpreting the user input (movement of two colored squares on the table) and translating the information into two mallet positions on our virtual representation of the air hockey table. The physics engine is responsible for taking in object positions (2 mallets and a puck) as input and detecting collisions with the puck. The physics engine then outputs the updated puck positions. The graphical interface is responsible for producing the two perspectives of the hockey table along with the objects on it. In addition, the graphics interface displays any relevant information about the game state, such as the current score, when the game is paused, etc. To accomplish this, the graphical interface takes in information about the game state and object positions. Finally, the module that glues the other components together is the game logic. The game logic component keeps track of current game state information (scores, etc.) and handles IO signals (pause/restart/replay, etc.).

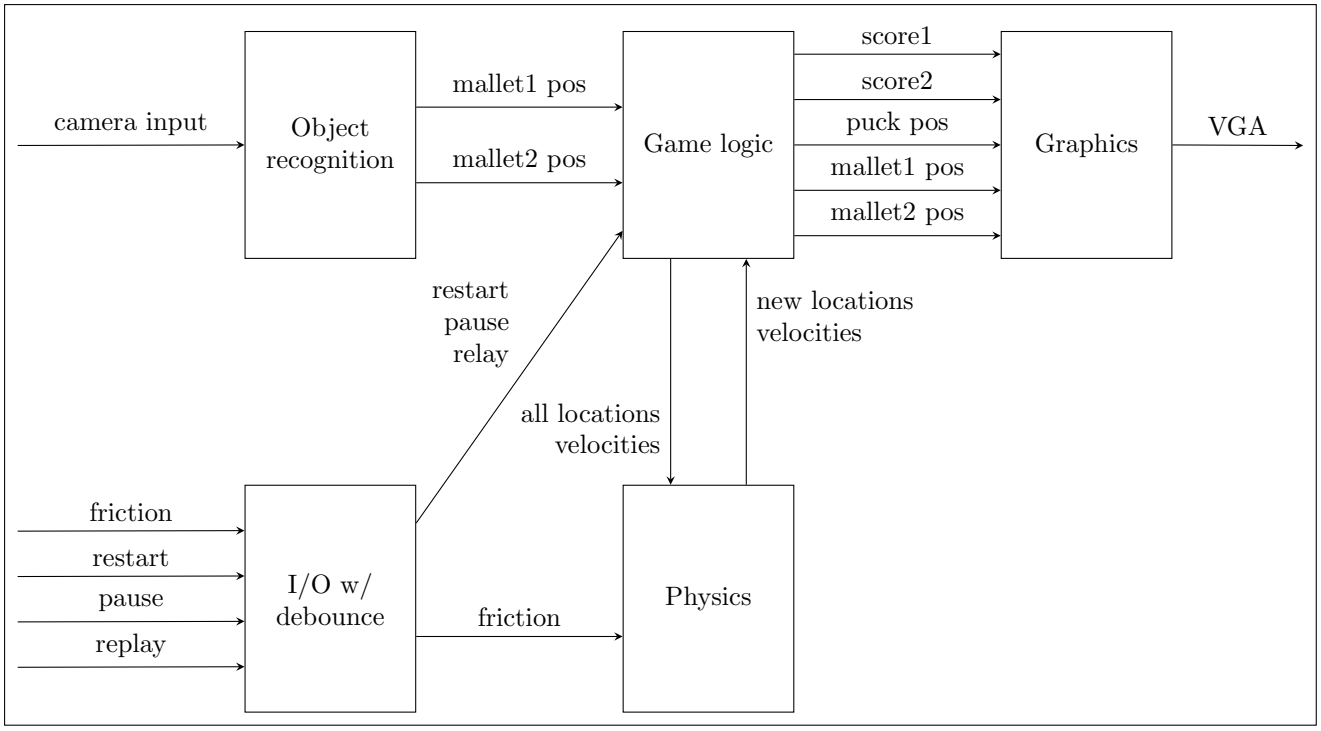

Figure 1: The overall block diagram features four modules. The object recognition module converts a camera input into mallet positions, which are then fed into the game logic module. The game logic module keeps track of the object positions and player scores. Object positions are updated with help from the physics module, and the game is displayed using the graphics module. The fifth module in the bottom left is just for debouncing input and output and is relatively minor.

Yuqing was responsible for the object recognition module. Abe was responsible for the physics module and the game logic module. Alex was responsible for the graphical interface. Lastly, we were all responsible for the integration of the 4 modules.

3 Modules

3.1 Object Recognition (Yuqing)

3.1.1 Overview

First we will give a high-level overview of the object recognition component, shown in Figure 2.

The purpose of this component is to process the NTSC video stream from the camera as input and ultimately determine the positions of the two mallets on the virtual air hockey table. Figure 3 below shows the physical setup of the mallet controls recorded by the camera.

The object recognition process can be generally organized into two separate stages. The first stage involves transforming the input data from the camera into a useful form. Next, in the second stage we use this modified camera data to generate the positions of the two mallets on our air hockey table. Our virtual table is a 1024 x 512 rectangle on which Player 1 stands from the left and Player 2 stands from the right. Figure 4 demonstrates a sample output of the object recognition component corresponding to an input in Figure 3. The outputted values are the coordinates of the two mallets.

3.1.2 Module Details

In the first stage of object recognition, we process the camera input into a more readily usable form. The module reads in NTSC video data from the camera as a YCrCb color stream using the

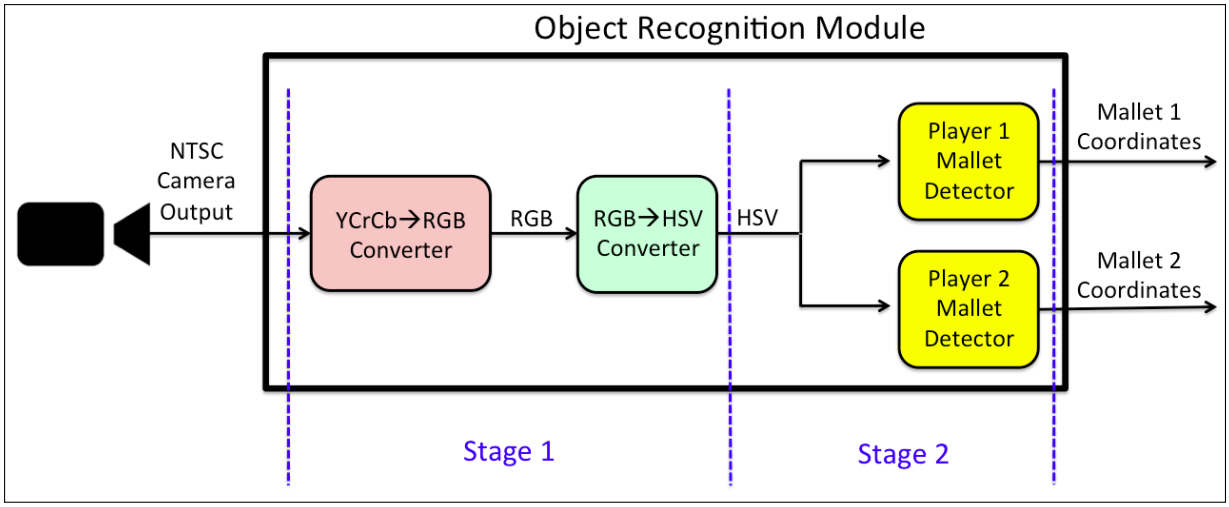

Figure 2: The object recognition module. This module interprets the camera input and produces two mallet positions on the virtual air hockey table.

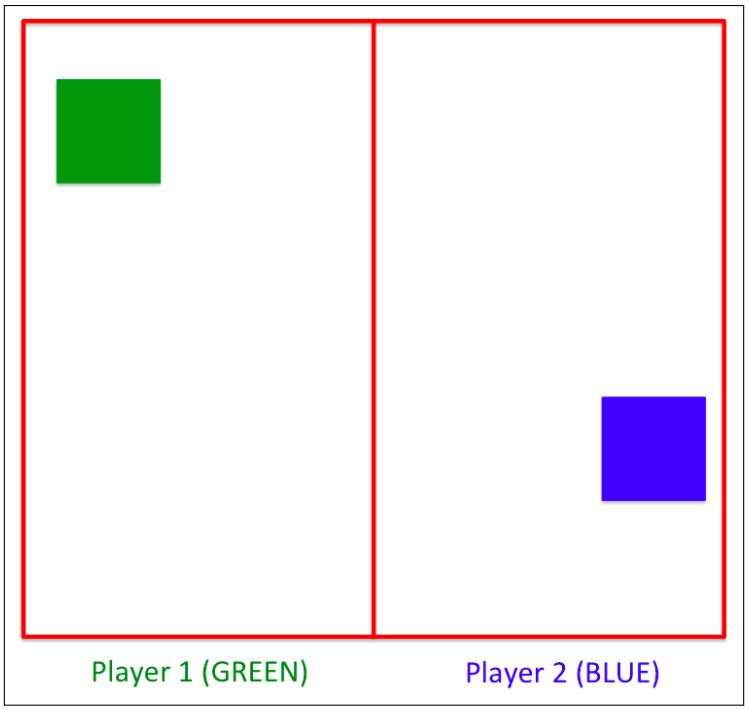

Figure 3: Object recognition setup. The two colored squares function as mallet controls. Players can slide the mallet controls anywhere in their allocated half of the legal playing field. Boundaries of the legal playing field are in red.

staff-provided ntsc_code module. Then, we convert the YCrCb data into RGB format and store the result in ZBT memory, which we implemented by modifying the staff-provided ntsc_to_zbt module that stores grayscale data. Finally, we read the RGB data from ZBT memory and convert it to HSV format, which more readily facilitates color detection. Detecting color in HSV requires us to check only the hue (H) bits, while in RGB format we would need to check all of the RGB bits. This allows more convenient calibration. Furthermore, isolation of the hue component possibly allows for more resilient color detection.

The second stage of object recognition is more involved. In this stage, we take the HSV color data corresponding to our color input and produce the coordinates of the two mallets on the virtual table. Our system involves two mallet detectors, one for each player. The

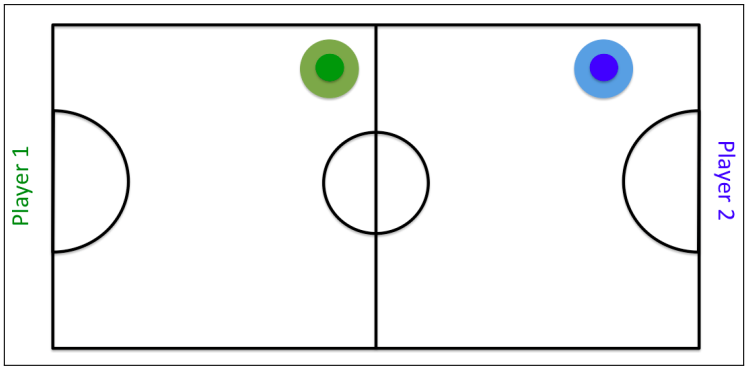

Figure 4: Sample results of object recognition. The mallet controls are interpreted as if the players were standing on opposite sides of the table.

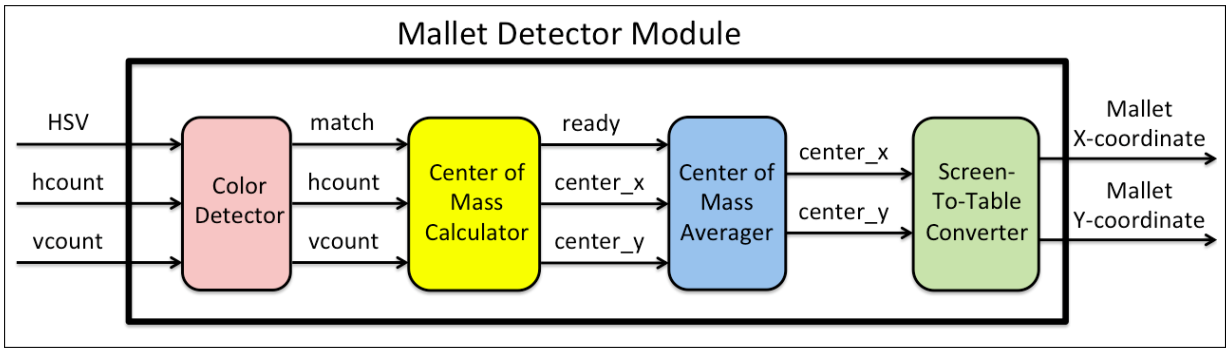

Figure 5: Mallet detector module. The mallet detection process consists of a few steps.

Mallet Detector modules receive a stream of VGA pixel coordinates, hcount and vcount, along with their corresponding HSV data. Each Mallet Detector outputs a single mallet position (corresponding to player 1 or player 2). Figure 5 summarizes the parts of the Mallet Detector module. We omit some details and rename some modules for clarity.

Inside the mallet detector, we first have a Color Detector module that sends a high "match" signal whenever the inputted hue value is within a target range. For player 1 this target range is a shade of green, and for player 2 this target range is a shade of blue. This signal is used by the center of mass calculator, which calculates the center of mass of a given color range within a single frame.

The center of mass calculator module consists of three accumulators and two dividers. The center of mass is calculated by averaging the x-coordinates and y-coordinates of all pixels that fall in the color range (i.e. for which the "match" signal from the Color Detector is high). To do this, we keep separate sums for the x-coordinates and y-coordinates, and we keep a count of the number of pixels of the target color. We then use a Core Gen divider to compute the averages. In computing this quotient, one must be careful about selecting a proper divider size. Selecting a divider too small produces overflow errors, while selecting a divider too large unnecessarily hinders performance (a larger divider lengthens compilation times and increases latencies). The camera produces approximately 500 x 700 pixels – if a detected object occupies a sizeable portion of the screen, then the coordinate sums become quite large. One other detail in the center_calculator module is identifying start and end of frames. In our implementation, we use the pixel coordinates of the top left and bottom right corners of the camera displays as these indicators, Each time hcount and vcount reach the top left corner of the camera display, we have begun a new frame, and we clear the accumulators. Each time hcount and vcount reach the bottom right, we have finished the frame, and we wait for the outputs from the divider. When the quotients are valid, we output them and send a high "ready" signal.

Although these center of mass positions could be directly translated into positions on the virtual hockey table, calculations from a single frame are relatively more prone to noisy behavior. Our solution to this problem is to smoothen the signal by maintaining the average of the last 8 positions (effectively sending the signal through a low pass filter). The Center Of Mass Averager module implements this smoothing.

Finally, the Screen-To-Table Converter module translates the smoothed center of mass positions into positions on the virtual air hockey table. Our internal representation of the table is a 1024 x 512 rectangle, as shown earlier in Figure 4. The mallet control is configured as if Player 1 is on the left side of the table and Player 2 is on the right.

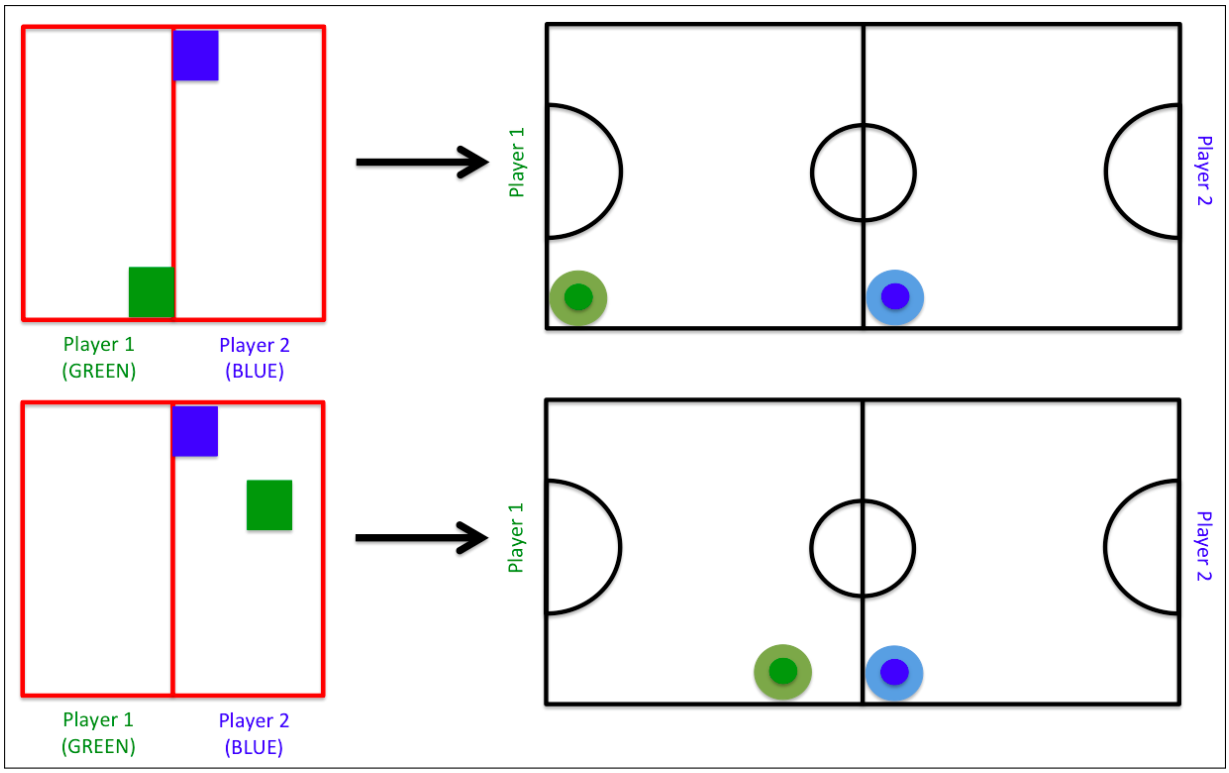

Figure 6: Example coordinate mappings. Example mappings from physical mallet controls to the mallet positions on the table. Notice that in the second example Player 1's mallet control is in the wrong half of the game field. The position of the green control is interpreted as on the boundary.

This change of coordinates consists of a rotation (in opposite directions for the two players) followed by a linear scaling. In our implementation, each player can move the puck only within his own half of the table. One surprisingly time-consuming aspect of the mallet position generation was allowing the players to easily maneuver their mallets across the entire legal table, the table edges in particular. The players should be able to move the mallets against the walls, even though the center of mass of an object will never hit the physical game field boundaries. (If the entire colored mallet control is in bounds then its center of mass must be inside the bounds as well). Our solution involves establishing a smaller legal center of mass field for the center of masses. We generate the table positions from the smaller center of mass field, instead of the entire game field. We manually optimized the parameters of the center of mass field to allow for comprehensive movement, making the field small enough for a player to be able to cover the entire table space. Meanwhile, too small of a center of mass field creates oversensitivity to small player movements. More details about the implementation are presented in the Implementation Process section. Figure 6 below shows example mappings from the physical space into representations on the virtual table.

One additional feature we implemented in this transformation is to disallow the players from

interfering with other player's half of the table. When a player places his mallet control in the other player's territory, it is read as if the mallet control is on the center boundary.

3.1.3 Implementation Process and Testing

Overall, the implementation of the object recognition was successful. When finished, the component allowed players to move their mallets to any legal region of the table with fine control. The averaging virtually eliminated arbitrary movements caused by noise. Testing was done mostly by using the VGA display and the hex display on the labkit. A main obstacle in implementation was the compile time. Every compilation required 30-40 minutes, seriously bottlenecking the rate at which we could test new features.

Figure 7: Color video successful.

The first step of building this component was to modify the staff-provided camera modules to work in color instead of grayscale. This process was significantly more involved than we expected, as we had underestimated the number of pieces involved in writing the video data to ZBT memory and reading it back. In retrospect, we would have saved a significant amount of time by choosing to seek more guidance. Instead, we spent a large amount of effort in attempting to understand the code and identifying the necessary changes. In the end, however, we were able to handle the colored video data, as shown in Figure 7.

The next step was to identify objects of a particular color. To do this we converted the RGB video data to HSV format, and checked to see if the hue (H) bits were within a target range. We tested the detection by coloring over the recognized objects. Initially, we made a naive mistake of matching color by checking the high order bits. This method introduced an unintended bias: consider the hue 0x0F. Matching the high order bits selects for the colors with hue values slightly less than 0x0F – the hue values slightly larger than 0x0F are ignored. We later resolved this issue by checking if the absolute difference of a detected hue and a target hue was within a specific bound. To calibrate our system, we generated crosshairs on the VGA display and printed to the hex display the hue value of the pixel in the crosshairs, as shown in Figure 8 below.

Once we were confident that the color detection was functional, we proceeded to calculating the centers of mass. The process through which the centers were calculated are described above in the Module Details section. To verify correctness, we displayed crosshairs corresponding to the calculated centers of masses, as shown in Figure 9. The performance of the smoothing (averaging) was tested in the same manner with crosshairs.

Figure 8: Crosshairs example. Pixels within a range of a target hue were colored over to signal detection. The hue value in the crosshairs was outputted to the hex display for calibration.

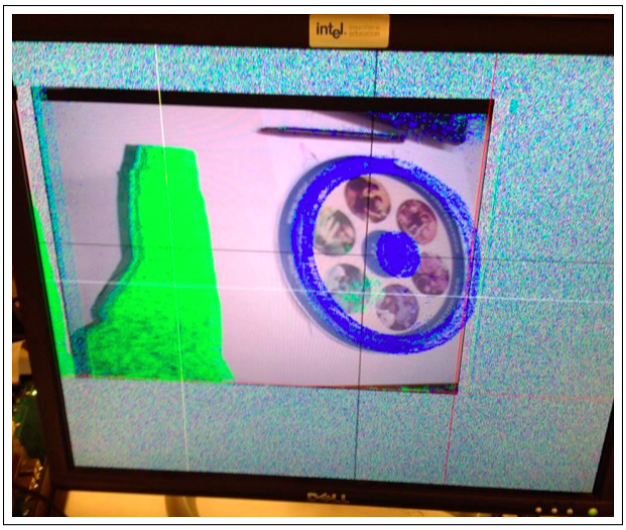

Figure 9: More crosshairs. White crosshairs intersect at the center of mass of the detected green regions, while the black crosshairs intersect at the center of mass of the detected blue regions.

We originally encountered overflow issues. Calculating center of mass requires computing an average of all pixels of a detected color. This involves computing a sum of all relevant coordinates. Since the camera display is approximately 500 x 700 pixels, a sizeable object produces large sum. After increasing the number of bits in our sum accumulators and correspondingly increasing the size of our divider module, the overflow issues disappeared and the center of mass was behaving as expected. An alternative solution to the overflow problem would be to truncate low-order bits. However, this reduces the accuracy of the resultant center of mass, so we did not take this alternative approach.

A second major challenge of the center of mass calculation was dealing with camera noise. Our particular camera seemed to consistently produce noisy outputs on the edges and in the corners, and this noise created enough false-positive detected pixels to hamper the accuracy of the center of mass calculation. Our solution to this problem was to disregard the edges and corners of the camera display in calculating the center of mass. Only pixels within an inner rectangle would be allowed to contribute to the center of mass, as shown in Figure 10. The

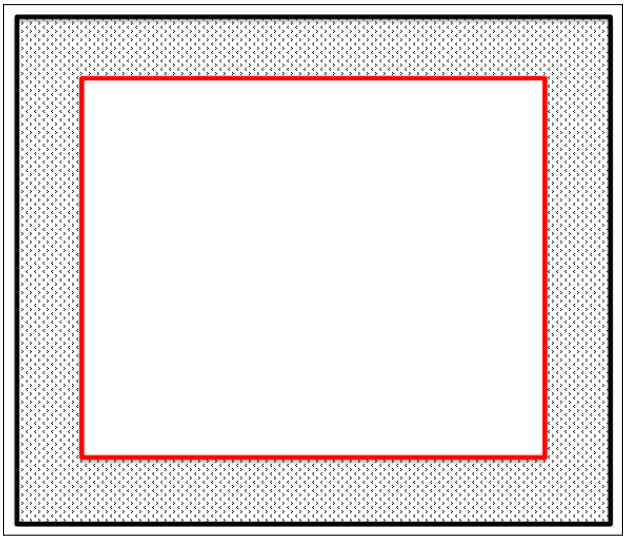

Figure 10: Camera borders. The black rectangule represents the entire camera display. To disregard the heavy noise around the boundary, only pixels within the red rectangle were used in calculating the center of mass.

boundaries of the physical game field were drawn to match the inner red rectangle.

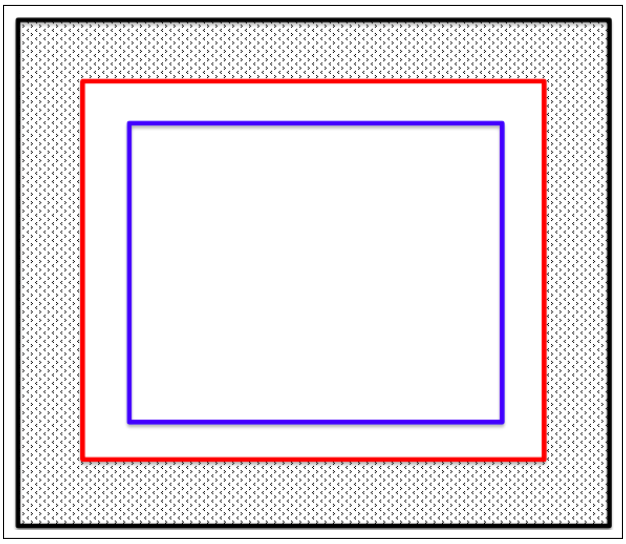

Figure 11: Inner camera borders. Instead of mapping the red rectangle to the entire virtual table, we mapped the blue rectangle. This way, the players could move the center of mass to the boundary of the mapping.

The final step was to convert the averaged center of mass to a coordinate on the table. As described in the Module Details section, the transformation consists of a rotation and a linear scaling. One of the larger implementation struggles was to guarantee that both players would be able to reach the entirety of their half of the table. We originally made the mistake of mapping the entire legal game field (the red rectangle in Figure 10) to the virtual table. This actually made it impossible for the mallets to hit the edges of the table, since the center of mass must always lie strictly inside the red rectangle. We resolved this issue by instead mapping a smaller center of mass field to the table, as shown in Figure 11.

Finally, taking care of all the details required multiple attempts. For example, we must properly handle the cases in which the center of mass lies outside of the blue rectangle. We must also recognize that the mallets themselves are sized – only the mallet boundaries, and not

the edges, can be tangent to the table edges. In addition, as described in the Module Details section we implemented a feature to disallow the players from interfering with the other half of the game field. This introduced some special cases as well.

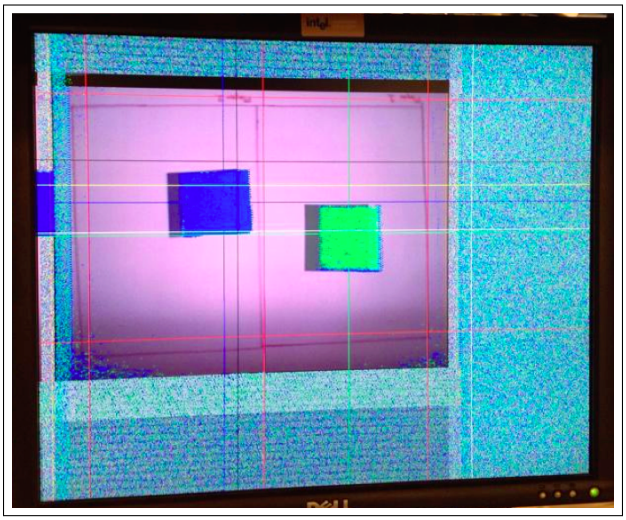

Figure 12: The "Calibration Mode" display. We see the detected blue and green regions colored in. The blue and green crosshairs correspond to the smoothed centers of mass, while the white and black crosshairs correspond to positions on the virtual table following the coordinate conversion.

Ideally, the cleanest method of testing the coordinate conversion would have been to connect the output to a functional graphical interface. Since the graphical interface was not yet available, we instead tested the coordinate conversion using crosshairs, shown below in Figure 12. We drew crosshairs corresponding to the resultant mallet positions. This was slightly confusing to look at, since we now had crosshairs corresponding to both physical and virtual objects on the same screen. In the end, the testing was relatively effective. We preserved this final testing display as a "camera calibration mode" for the game itself. Before playing the game, the players can use this screen to verify that the camera has been calibrated correctly.

3.1.4 Review and Recommendations

The object recognition component was generally very successful. It presented players with smooth and very fine mallet control throughout the entire intended half of the table. A major reason for the success is that we did not underestimate the time required to fully debug the system. Since every compilation of the objection recognition component required 30-40 minutes, we were allowed relatively few iterations of the product. Anticipating unexpected issues, the team member responsible for this module cleared out the last week before the checkoff (from other coursework and activities) to dedicate to debugging and polishing the object recognition system. As a result, the object recognition component was mostly finalized a few days before the checkoff. This allowed the team member to spend most of the last few days managing the integration and helping with the other modules.

Another key factor in controlling the debugging time is writing clean code. Because of the long compile times, messy code becomes very costly – simple, elusive bugs waste many iterations of the development process. In hindsight, we were fortunate that we had invested the time to produce well-organized code – this made mistakes much easier to catch later on.

One aspect in which we could have improved in was utilizing our resources. We might have attempted to complete our work a bit too independently, when asking for guidance from the TAs and instructors might have saved us a large amount of time. For example, understanding and modifying the staff-provided ntsc_to_zbt might have consumed much less time if we had asked for a bit of guidance. Similarly, we had a very brief conversation with Luis about the high

level approach of object detection. We likely would have been much more efficient if we had requested more involved discussions about our implementation details or about our methods for handling camera noise.

3.2 Physics

3.2.1 Overview

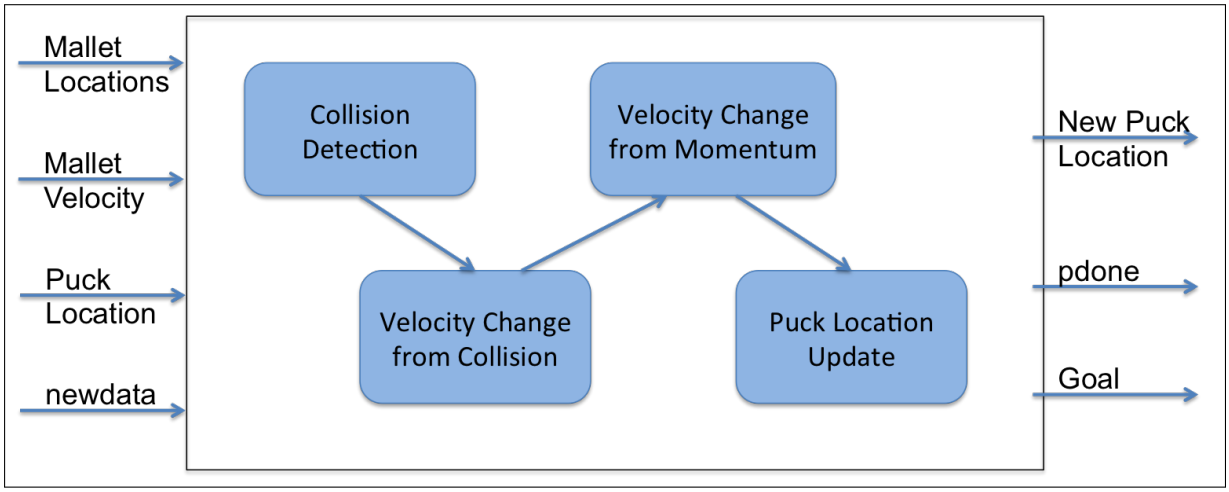

Figure 13: The physics module. The physics module handles calculating new object locations and velocities given old ones. In some cases the input variable name equals the output variable name. These represent cases where the variable is not modified and is fed directly out of the module. One of the outputs of the collision detection module is a boolean denoting whether a collision occurred.

The physics engine module, shown in Figure 13 will be responsible for calculating the projected velocity and location of the puck based on the current velocity and locations of the objects. The goal during the design phase of the project was to have the collision of the puck against the walls and mallets determine the new velocity and locations of the puck. The extended goal was to change the velocity and location of the puck also by considering the friction between the puck and the table. During the time given, we were able to meet the intended goals, but could not find a way to implement friction on time. More on this will be discussed in a later section.

This module receives all the necessary information from game logic module and, upon completion of all the calculations, sends the updated data back to the game logic module. Therefore, the communication between the game logic module and the physics module is as important as ensuring that the physics module functions as expected. Since it takes several cycles for the physics module to calculate its result, we created two variables, "pdone" (short for "physics done") and "newdata" (short for "new data to be sent"). "Newdata" will be set to 1 whenever the game logic module is ready to send all the necessary data to the physics module. "Pdone" is set to 1 when the physics module has finished calculating the new location and velocity of the puck. Both variables will be set to 0 in the clock cycle following the cycle where either of the variables are set to 1. We paid extra attention to this part in coding to make sure that the game logic module sets "newdata" only after "pdone" is set by the physics module, and vice versa.

The actual internal workings of the physics module is divided into 4 different states: standby state(0), collision state(1), friction state(2), and send state(3). In the standby state, the module waits for a "newdata" to be set by the game logic module. Once set, the physics module enters collision state, during which the module determines any collision between the puck and other objects and adjust the new velocity accordingly to the laws of collision. Then, it enters the

friction state, which is connected to a switch that determines whether the puck will be slowing down as it travels. Once all updates to the new puck velocity is done, the module enters send state where it sends back the new puck location to the game logic module.

3.2.2 Module Details

3.2.2.1 Collision against the Wall As the physics module enters the collision state, it will immediately check whether the puck is near a wall. Determining the new velocity after the collision is very simple: if the puck hits the top or bottom wall, its y-directional velocity switches its sign while the x-directional velocity changes its sign when the puck hits the left or right wall. The module, however, must be careful of the case when the puck enters the goal region on the wall, during which case the velocity of the puck must be set to 0 to keep it in the goal.

The tricky part of wall collision is determining exactly when the puck hits the wall. We came up with two separate schemes for this detection. The first method examines the puck's current location and determines if the puck is sufficiently close to the wall. In this case, the walls would be padded with extra bits, or "pads", so that the puck is in collision with the wall if the distance between the puck and a wall is less than (radius + pad). The second way predicts where the the puck will be in the next cycle if the puck did not collide with anything in the current cycle, then declare collision if the next location is outside the coordinates for the table. We determined pros and cons for each implementation by testing both schemes in a simulation.

The first case was quite simple to implement and did not take many lines. It reliably detected wall collision as long as the puck was not moving so fast that the puck would simply travel through the padding in one cycle. The anticipated draw back was seeing the effect of padding on the screen by having the puck change its velocity far from the wall, but the simulation showed that there wasn't much noticeable gap between the wall and where the puck bounces. The second case did not go "through" the wall and appeared on the other side of the field as in the first case in high speed, but the simulation showed a noticeable gap between the wall and when the puck bounced. In certain conditions, the puck ended up going through another object, such as a mallet or a different wall. Since this second case led to more problems in simulation, the first scheme was implemented for all collision detection.

3.2.2.2 Collision against Mallets Calculating the effects of the collision between the puck and users' mallets are more complicated than the one against the wall. Unlike the walls, which are flat surfaces, the mallets are circular objects. If the puck bounces off of a mallet at an angle normal to the circumference, the resulting velocities would only change in their signs. If the puck collides the circumference at an angle, however, we need to calculate the angle of reflection/incidence and change the direction of the resulting velocity accordingly, which requires using two trigonometric expressions: one to determine the angle of the point on mallet at which the collision happens and another to determine the angle of reflection/incidence. Although this was complicated to code, I still implemented this method to see if it was possible.

The biggest challenge here was emulating a trigonometric equations. Since we could not find a way to implement sine and cosine functions using circuits, we instead calculated the values of the functions corresponding to different angles from 0 to 2π in Matlab and created a look-up table. For testing purposes, we created 24 samples from $\frac{\pi}{12}$ to $\frac{24\pi}{12}$. This method was unsuccessful because the module often had trouble pinpointing exactly where on the circumference of the mallet the collision had occurred and preserving the magnitude of the puck's velocity throughout the process created too much latency due to the amount of multiplications and divisions that needed to be done.

Instead, we modeled the mallet as a simple octagon, as shown in the figure below. The octagon is sized so that the sides that are parallel to the walls are shorter in length compared

to the other sides to make it interact more as a circle than a regular octagon. Some examples shown in Figure 14 shows how the mallet's direction will change after the impact.

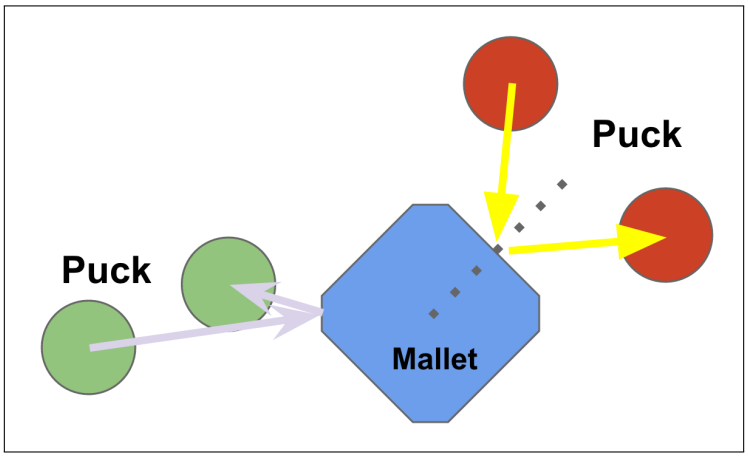

Figure 14: Collision example.

Another important phenomenon we paid attention to with mallet collisions is the momentum transfer. In an actual air hockey game, players move the mallet towards the puck to make the puck travel faster after the impact, and we wanted to emulate this behavior. Our initial implementation was to detect if there is mallet collision and then add half of the mallet's velocity to the puck's velocity. This scheme proved to be problematic when the mallet or the puck was traveling at a high speed during an impact. When the objects are moving at high speed relative to each other, the mallet and the puck are detected to be colliding multiple times during a collision since it is possible for the objects' coordinates to be overlapping after a cycle. When this occurred, the puck's velocity changed too rapidly, so as for it to come to a halt or starting traveling so fast that the pucks would be seen in continuous motion on the screen.

After the checkoff, however, we found a way around this problem by transferring the mallet's momentum only after the puck and the mallet have stopped colliding. This way, even if the objects were to "collide" multiple times during one impact, the mallet's velocity affects the puck only once. Further, we capped the maximum velocity possible to be contributed to the puck. Since the object recognition module can be noisy at times, the velocity of the mallet may be calculated to be much bigger than it should be, creating an unrealistic momentum transfer. After limiting the change in velocity due to momentum, the puck's interaction with the mallets looked much smoother and realistic in the simulation and the final product. We arrived at the point where one could use the mallet to slow down or speed up the puck using the momentum.

3.2.2.3 Incorporating Friction Providing an option to slow the puck down as it travels was an extended goal. Since we had completed the implementation and simulation of the basic physics module before Thanksgiving, we decided to tackle this problem. The very first method was to decrement the new velocity by a small fraction if the puck did not have any collision in that cycle. To avoid using dividers, we simply shifted both the x-velocity and y-velocity of the puck by a certain amount and subtracted that from the original velocity.

The problem with this implementation came from the fact that we were dealing with digital circuits. All the velocity values, as well as any other variables, are represented as a string of 1's and 0's, so using shift-and-subtract on big numbers worked reasonably well, while the method failed to slow down already small numbers (slow puck). For example, the initial velocity of the puck was 10 in x-direction and 20 in y-direction and the friction shifted each velocity values by 2 each time:

$$1010_2, 10100_2 \rightarrow 1000_2, 1111_2 \rightarrow 110_2, 1100_2 \rightarrow 101_2, 1001_2 \rightarrow 100_2, 111_2$$

As shown above, the proportionality of the velocity in x-direction and y-direction are not preserved with each cycle. Therefore, the above implementation made the puck change the course of its trajectory every time shift-and-subtract was done. Since we could not find a way to preserve the direction of the velocity while accounting for friction without representing the velocity with a lot of number of bits, this extension was not implemented in our final design.

3.2.2.4 Updating New Positions After calculating the new puck velocity after the collision state and friction state (which does not have any effect on the velocity due to the lack of its functionality), the module arrives at the send state, during which the module must determine the next puck location based on the current location and the newly calculated velocity. During the same cycle, the state sets "pdone" so that the game logic module can finally accept the information being output by the physics module.

3.2.3 Implementation Process and Testing

A lot of difficulties we faced while designing and testing this module, such as collision detection, circular collision, and communication between the physics and game logic module, was predicted from the beginning of the process, but the most annoying and persistent challenge we faced was variable treatment.

Resolving the conflict between signed and unsigned variables took the most time to debug. We knew that all the positions will be represented as unsigned numbers while all the velocities will be represented as signed numbers, but we made many assumptions about how operations work with the two kinds of numbers. For example, when calculating the new puck position using the puck's old location, an unsigned number, and the puck's new velocity, a signed number, we thought we could simply add the two numbers together, and verilog would somehow make the numbers unsigned in the end. As it turned out whenever there is an operation between an unsigned and a signed number, the result is usually a signed number or sometimes ambiguous. Because we did not expect this to be the source of many odd behaviors observed during simulation, it took a long time to track and fix these bugs by forcing them into being signed or unsigned numbers using the appropriate procedures in verilog.

Other similar bugs arose when we started doing operations on numbers with different number of bits and many other. After fixing the first few bugs, we had become more familiar with how to track down the sources of bugs not only from typos and accidental commenting but also from the conflicts in the nature of variables.

Testing of this module was done by replacing the physics part of 6.111 Lab 3, during which we designed a pong game. In this testing simulation, we can control the movement of the mallet using the up, down, left, and right buttons on the labkit. We changed the dimension and goal areas of the field to reflect our air hockey dimension of 1:2. During the debugging process, we checked various values, such as puck velocity and distance between objects, by connecting those outputs to the display and LEDs on the labkit.

3.2.4 Review and Recommendations

The physics module worked well as long as the velocity of the puck did not exceed about 15 pixels by cycle. After the specified speed, the speed of the puck was too fast to be seen in smooth motion on the graphics. Further, the high speed puck starts going through walls and objects. The only way to fix this is to use higher definition and the faster clock to make the pixels per cycle smaller for the same velocity. If we were to do a similar project in the future, we would pay more attention to the declaration of each wires and registers to make sure than any operations I do with them don't produce unexpected outcomes.

If given more time to work on the module, we would try to find a way to implement friction and angular momentum in the physics module. As discussed above, we have not found a way to

decrease the velocity of the puck in both x and y direction while preserving the direction, but this slowing down motion is often seen in many video game applications, so we may be able to find a way to implement friction if we conduct a research on how this is done in other publicly available games.

In a real-life air hockey game, avid players utilize angular momentum in various ways to manipulate the trajectory of the puck after the collision following the mallet hit. Our model disregarded any angular velocity so that the angle of incidence and the angle of reflection are the same, but finding a way to determine and use the angular velocity information of the puck to calculate the change in direction after the collision may be a challenging yet exciting problem to tackle.

3.3 Graphical Interface (Alex)

3.3.1 Overview

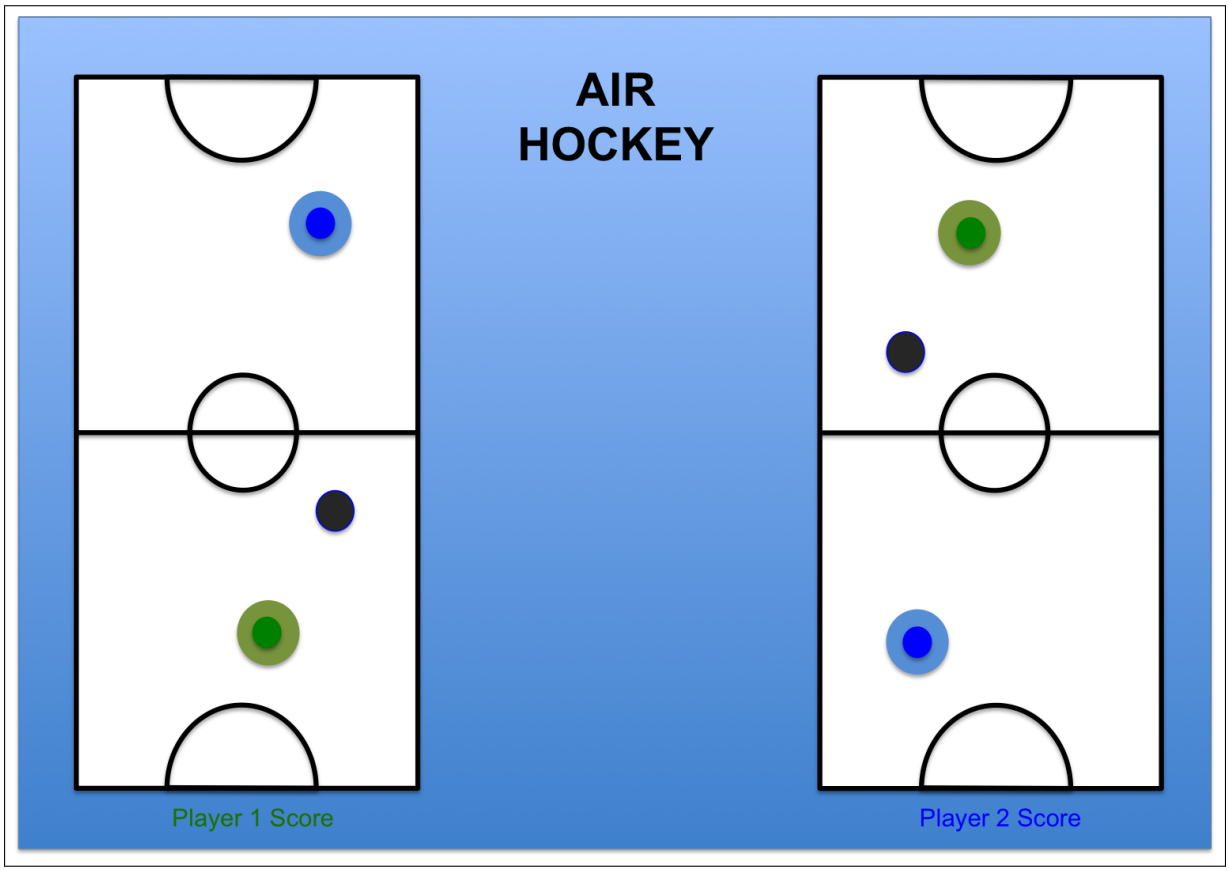

Figure 15: A mockup of what the air hockey graphical interface is supposed to look like. Player 1's view is on the left, and player 2's view is on the right. The blue and green objects represent the mallets and the black circle represents the puck. The white rectangles are the tables and they feature decorations, depicting the midway line and the goals.

The primary purpose of the graphical interface module is to convert information about the air hockey game into something displayable on a monitor (resolution 1024x768) through VGA. Internally, the game is represented on a table with width 1024 and height 512, and the game objects, the mallets and the puck, are circles. The additional information that needs to be displayed are the player scores as well as messages that come with the current game state. The goal is to display this information as cleanly and as elegantly as possible. A mock of the target layout is shown in Figure 15. The actual implementation looks like the one in Figure 16.

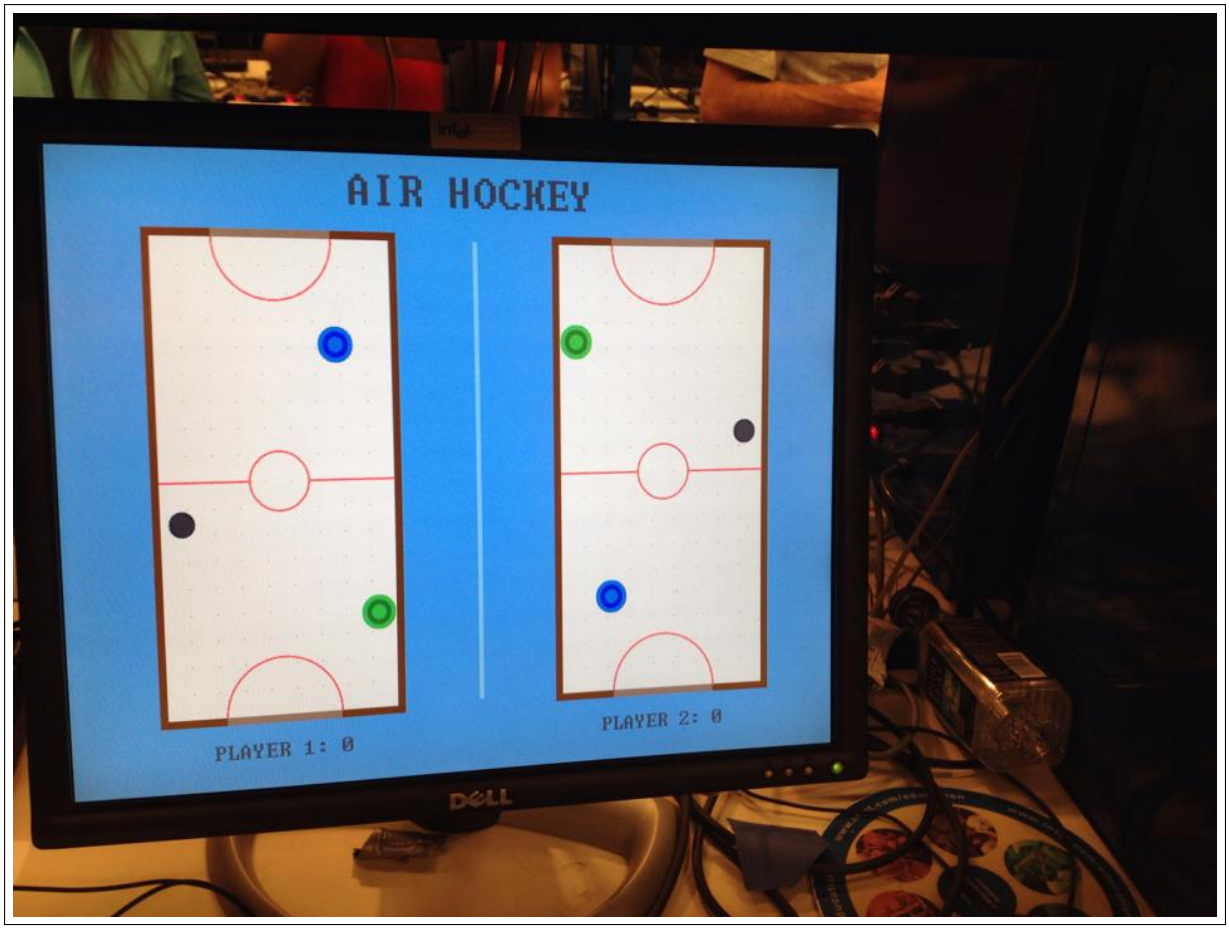

Figure 16: Our 2-dimensional rendering of the air hockey game. If you look closely, you can see the array of dots on the air hockey table.

In addition, in order for both players to get a good view of the game, the graphical interface should render the air hockey table from two perspectives, one from each player. This way both players have a table to look at, as if they were actually playing the game against each other in real life.

As a note, the original goal of this project was to implement the graphical interface to provide a 3-dimensional view of the air hockey table. However, during the implementation phase we decided to just render the 2-dimensional version. This decision process is described further in the "Implementation Process (addendum)" section.

Figure 17 shows the block diagram for the graphics module.

3.3.2 Module Details

For an idea of what the resulting graphical interface looked like, see Figure 16. On each side of the middle divider is a view of the air hockey table from the perspective of each player. In addition, Figure 18 shows what happens when the game is paused and what happens when a goal is scored. In both cases, text is overlaid on the screen along with possibly other modifications.

The graphical interface was contained in standalone module called graphics2d. This module took as input the locations of the 3 game objects, both player scores, and information from the xvga module used to interface with the VGA. During each clock cycle, the graphics2d was given a location of a pixel on the screen and the module had to output the color to be assigned to that pixel. For a 60Hz refresh rate on a 1024x768 pixel display, we used a clock cycle of 65MHz and aimed to generate one pixel per clock cycle (with a throughput of 1 pixel per clock cycle).

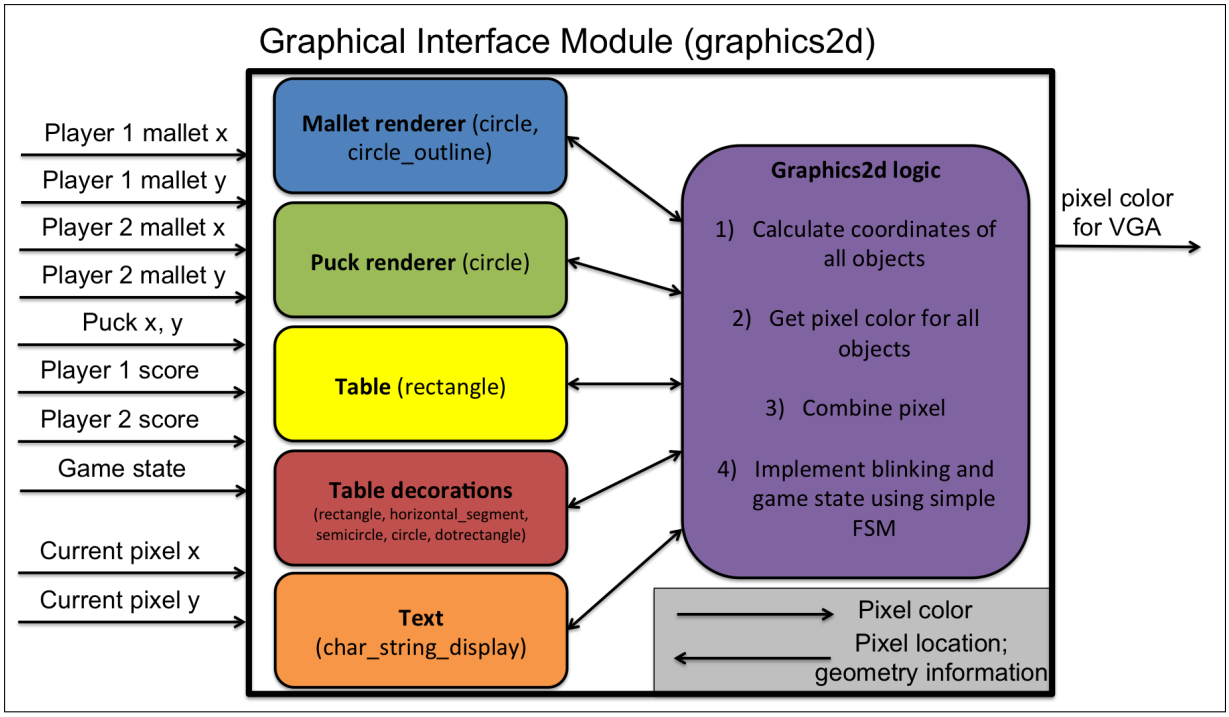

Figure 17: Block diagrams for graphics. The graphical interface module accepts player scores, object positions, other game information (whether the game is paused or in replay mode), and a current pixel location as input and outputs to the color that the pixel should be. The main logic uses several groups of modules (on the left) that each accept a pixel position and a geometric object and output the color of the geometric object at that location. These pixels are then combined to decide the finalized pixel.

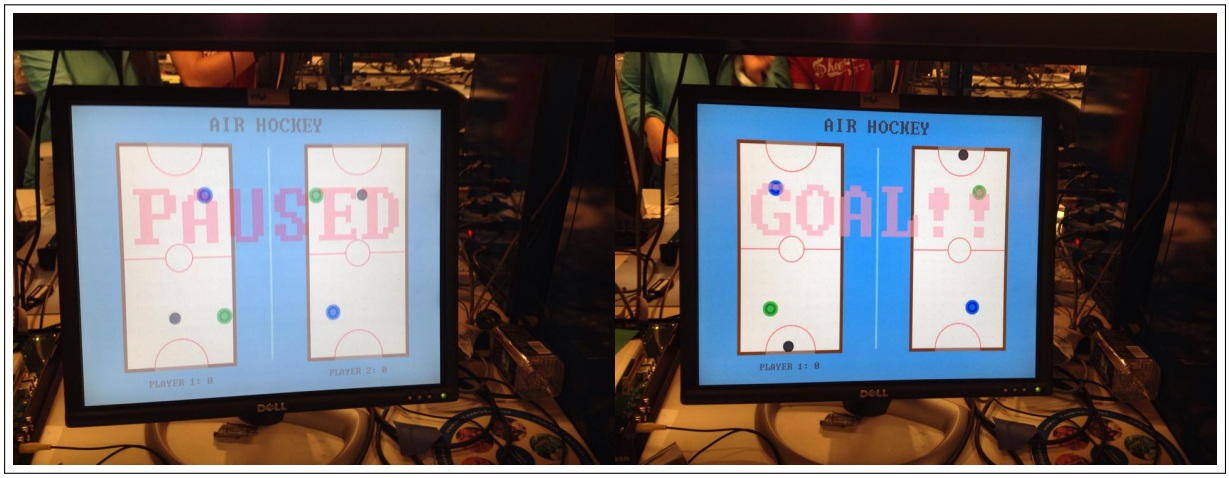

Figure 18: Two examples of the text overlay for the air hockey game. In the left, the player score flashes when a goal is scored. On the right, when the game is paused, the rest of the screen is alpha-blended with a white rectangle.

It would be very difficult to use this model for graphics if the throughput was less than 1 pixel (or 1/2 of a pixel) per clock cycle, so we tried to keep pixel generation fast and pipelined.

The graphics2d module was implemented in 3 sections. In the first part, various parameters were specified, such as the table size and the window size. In addition, many additional parameters were calculated from these parameters. For example, we had to compute the coordinates for both rectangular tables, as well as the dimensions of the rectangles and the game objects (mallets and pucks) to display on the screen, making sure that their relative sizes were consistent with that of the internal representation. The last part of these parameters included

specifying the coordinates of the centers of the mallets and pucks, which we computed via basic math.

Table 1: The list of graphical objects and the respective module used to generate the pixel values for the graphical object.

Graphical Object	**Description**	**Modules Used**
Text	The text includes the title, the player scores, and the overlaid "PAUSED", "REPLAY", and "GOAL!!" text.	char_string_display, char_string_display_2x, char_string_display_8x
Air hockey table	This was essentially a rectangle with a border.	rectangle
Dot array on the air hockey table	This created a grid of dots within a bounding rectangle.	dotrectangle
Goal indicators on the table border		rectangle
Mallets	The mallets were drawn as 3 concentric circles, similar to how one would look in real life when viewed from above.	circle, circle_outline
Puck	The puck consisted of two concentric circles.	circle
Table decorations	These were red markings that appeared on the air hockey table to mark the goals and the midway line.	horizontal_segment, semicircle, circle

In the second part of the graphics2d module, we computed a pixel value for each of the individual graphical objects on the screen. These objects included text, the air hockey table (rectangles), the game objects (2 mallets and a puck), the dot array on the air hockey table, and the table markings (goal lines and the midway line). Two copies of everything were rendered, one for each player. Each individual graphical object pixel was generated by some sort of module, as shown in Table 1. These modules all accepted a geometric object specification and a pixel location as input, and outputted the relevant pixel color at the location. If the pixel location was not on the object, then the output pixel would be black. We were careful to make sure that each pixel-coloring module took exactly 4 clock cycles to generate, and pipelined each of these modules so that one pixel color could be outputted during each clock cycle. For example, due to the multiplications required, we needed more than one clock cycles to calculate whether a pixel was within a circle. However, calculating whether a pixel was inside a rectangle could be done easily one clock cycle, so we used a synchronizer module to delay the output signal a number of clock cycles such that the latency would be exactly 4 clock cycles for each pixel to generate. The workings of each of the geometric object modules were relatively simple and just involved performing a pipelined computation to evaluate whether the input was contained within the geometric object. The text modules (which came in 3 different sizes) were adapted from 6.111 sample tools, and read text information from memory and outputted pixel colors accordingly.

In the final part of the graphics2d module, all the pixels generated from the second part were combined. Instead of somehow aggregating all the pixels, we ordered each pixel in terms of priority, and picked the highest priority pixel color that was not the default color (black), since black was used to specify a pixel that did not need to be colored. In the special case in which the game was paused, the pixels were alpha-blended with a white rectangle in order to give the effect that the game is behind some sort of a translucent screen.

The final part also contained a simple computation to make it obvious when a goal was scored. The graphics module detects when a goal is scored by tracking when the scores change. When this happens, the "GOAL!!" text is displayed on the screen for a small number of seconds, and the score that changed is blinked. The module kept track of the number of blinks remaining for each goal text, and every time a player score increased, the number of blinks was changed from 0 to something positive so that the player score text would flash. See Figure 18 for a sample image of this behavior.

3.3.3 Implementation Process and Testing

The implementation process consisted of using the VGA framework from the pong module and overhauling it to produce a more complex version for air hockey.

In order to get the 2-dimensional graphics working, we first performed all the mathematical calculations necessary to draw game objects in the correct locations with the correct sizes. Next, we converted all the game objects from rectangles to circles. This brought about a lot of incorrect pixels, due to both sign errors and the fact that computing whether a pixel lies in a circle might take more than one clock cycle. At this point, we incorporated the synchronizer module and modified both rectangle and circle to use four clock cycles but not cause any timing issues.

At this point, the graphics looked like it did in Figure 19. It was useful enough to be integrated but definitely nowhere near ready for presentation. Following that, we adapted the circle module to support semicircles and circle outlines, and from there we implemented the rest of the graphics. After handling all the geometries, we shifted focus to rendering text.

The graphical interface is very useful for testing the integrated module as well as the individual components. In order to actually test the graphical interface itself, we used the labkit and a monitor. As an input to the graphics module, we used the arrow keys to input mallet locations and very basic physics (from the pong lab) to control the movement of the puck. First, the modified pong game module was tested using the original pong graphics to ensure that no objects could leave the game boundaries (which are not the same as in pong). Once the pong game module was verified, we used it to test the implementation of the graphical interface.

To test the other inputs to the graphical interface, we used switches. Some of the switches would allow us to input the scores of each player, and other switches specified whether the game was paused or in replay mode. We made sure that only one of the textual elements ("paused" or "replay" or "goal scored") was ever displayed at once. For example, we tried to score a goal and then immediately pause the game, and we tried to pause the game while in replay mode. In both cases only one text object should be overlaid. It was also important to verify that the score blinking worked properly, even if both players just increased their score in a short period of time.

3.3.4 Review and Recommendations

One idea for this project that was not fully implemented but was suggested by the course instructors was the possibility of displaying the two air hockey tables on two different monitors. Then, each player would see his or her own perspective on just one screen, instead of both players having to share one monitor. This would improve the user interface. We implemented a version of the game that could take the game coordinates from the FPGA inputs rather than from the physics engine and render game objects at those coordinates, but did not integrate this. The transfer of game coordinate information could either be done in parallel by sending all 60 bits of coordinate information, 6 bits of player score information, and 2 bits of game state information from one FPGA to the other, but it could also have been done by using two connections and carefully executing a serial method of data transfer. In hindsight, this would

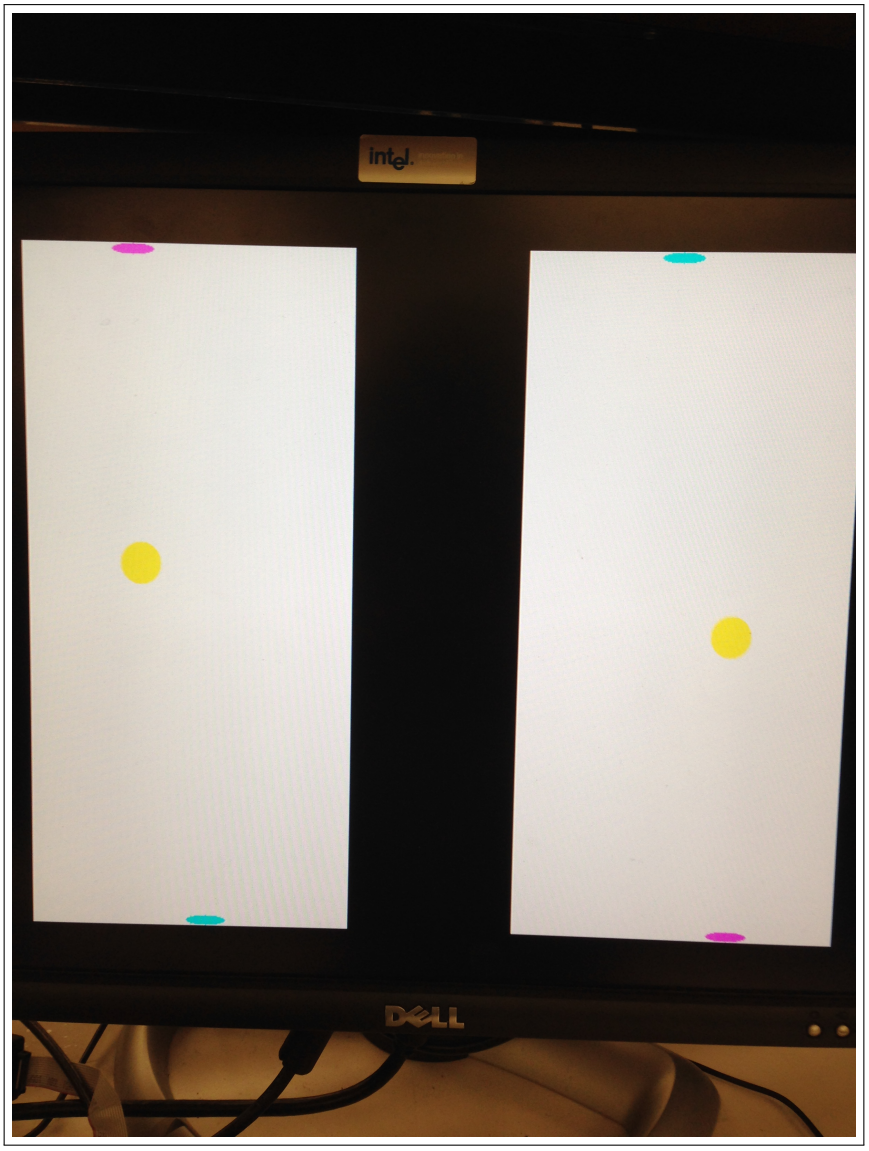

Figure 19: An incomplete 2-dimensional graphics rendering. The two tables are present, as are the mallets and puck. Elliptical objects were implemented, but not used in the final product.

have been a great addition to the game and more focus should have been given to implementing this.

The primary lessons from this module came from our failure to implement 3-dimensional graphics. In general, the attempt at 2-dimensional graphics was quite straightforward and went smoothly. However, the original goal was to implement 3-dimensional graphics, and that goal was not achieved. We underestimated the complexity of trying to display 3-dimensional graphical renderings of air hockey. We think it would have certainly been possible, especially if we had more time (started earlier) and experience. We created two different attempts at rendering 3-dimensional graphics but decided to scrap both of them in favor of 2-dimensional graphics. An unfortunate consequence was that a lot of time was lost on 3-dimensional graphics that could have instead been spent helping debug the integrated modules.

Seeking out more guidance from instructors would have greatly helped resolve problems more efficiently along the way and also would have helped us avoid losing time to things that would have never worked. In addition, the 3-dimensional graphics may have been possible with better planning at the very start, especially if we had given more thought to clock cycle requirements and FPGA resources.

3.3.5 Implementation Process (addendum)

In this section, which is mostly irrelevant to the actual 2-dimensional graphics that is now part of the game, we detail the implementation process for the 3-dimensional graphics. We tried several things but nothing to completion. We suspect that it was definitely doable, though.

3.3.5.1 Attempt 1: Ray Tracing From the start, a very basic 2-dimensional graphical interface with nothing but a rectangle and 3 circular game objects was created. From there, the goal was to take it to 3 dimensions. Initially, we went straight for the stretch goal of implementing ray tracing, believing that this would be quite doable. The general layout of the ray tracing module is as follows:

- For each table, decide on a location for the camera and the image plane. For each table there was a raytracer module, which draws a ray from the viewpoint (camera) location through each pixel on the image plane, such that every pixel on a 1024x768 rectangle on the image plane has a ray going through it.
- For each pixel on the image plane, find the intersection of the corresponding ray and every object that we wish to render (the table, two mallets, and the puck).
- Determine which object is the closest to the camera, and render the color of that object at the pixel location.

Initially, to simplify the model, we assumed the objects had no special refractive or reflective properties. That is, the rays never bounced off any objects, and we never considered any other light sources. This would have been far too difficult, and we hoped to approximate shading some other way. Following this plan, we implemented a ray collision module for both rectangular prisms (which represented the table) and cylinders (which represented the mallets and the puck). We also would have eventually implemented a collision module (ray and geometric object intersections) for spheres so that we could use spheres to depict parts of mallets, since a mallet can be rendered using a combination of spherical and cylindrical geometric objects.

In order to compute these intersections, we created a module to detect the intersection between a ray and a plane. Here, we ran into trouble because accurate ray tracing involved a lot of math, and it was difficult to make sure we could maintain a throughput of 1 pixel per clock cycle. The primary reason we never integrated the ray tracing 3-dimensional implementation is that we greatly underestimated how many pipelined dividers the implementation would require. Once we were pretty close to having the ray and plane intersection implemented we realized that there were not enough resources on the FPGA for all the dividers we wanted to use. We tried to simplify the system (such as by letting the table be just a rectangle), and we managed to avoid the usage of any square root modules, but this was not sufficient enough of a simplification because at least 6 cylinders had to be rendered.

We also considered using a frame buffer. That is, since the FPGA comes with two ZBT SRAM chips, we could use one to write pixel values to and one to read pixel values from. Every time we were done writing pixel values to one of the ZBTs, we would switch the roles of the two ZBT SRAMs. This would give us the freedom to compute the pixels with whatever speed we needed, as long as the display updated quickly enough to give a good representation of the game. The lack of timing constraints meant we could perform computations serially instead of all in parallel, and we would not have to use as many dividers. Unfortunately, the object recognition module already used one of the ZBT SRAM chips and this frame buffer idea would not have been possible. A very promising possibility that we didn't try was rendering the graphics on a different FPGA than the one we used for object recognition.

3.3.5.2 Attempt 2: Approximating 3D to 2D Projections The second main idea we implemented but later scrapped was to approximate the objects as various quadrilaterals and ellipses. A cylindrical mallet, when projected onto 2 dimensions, appears to be a combination of ellipses and quadrilaterals. We implemented ellipses as well as very approximate ways to map 3-dimensional coordinates onto a 2-dimensional planes. We could not be exact because this would require using sines and cosines as well as a lot more division than we could afford. Unfortunately, the approximations did not look very realistic. As a result, we did not complete this idea for 3-dimensional rendering. Perhaps with better approximations, this method could have worked.

3.3.5.3 Other Thoughts A much simpler implementation of 3-dimensional graphics could have been done by pre-storing images of the mallets and the puck in memory, and then simply rendering a scaled version of these images for the graphics. This was the original backup plan, but we did not go down this route, deciding it would be more interesting just to get a solid 2-dimensional rendering working. In addition, this would have been an approximation since we would have rendered a far away cylinder as simply a smaller version of the close-up cylinder.

These problems would have likely been avoided by thinking more about the design from the start and by the graphics implementer (Alex) having a better understanding of what is possible on FPGAs. With better planning and more time, it would have been likely possible to have gotten one of them working, since all the methods tried had actual workarounds and perhaps better approximations / optimizations that we missed out on (things to save resources or clock cycles). In the end it was disappointing to fall back onto 2-dimensional graphics but fortunately a lot was learned in the process.

3.4 Game Logic (Abe)

3.4.1 Overview

The game logic module is the main state machine module that communicates with all the other major modules. This module not only delivers important values from one module to another but also provides user interface. As described in the design of our project, it delivers the mallet locations received from the object recognition module to the physics module, waits for the newly updated puck location from the physics modules, and transfers all the necessary data to the graphics module. It also includes a BRAM memory to hold the past few seconds worth of objects' locations to be replayed later after a goal is scored.

The most important part of this module is relaying the data between other modules without fail or repetition. Therefore, a lot of time was spent coming up with the most efficient design for communication.

3.4.2 Module Details

3.4.2.1 Communicating with the Physics Module As described in the physics module above, "pdone" and "newdata" variables are used to communicate between the physics module and game logic module. While the game logic module operates on 65 MHz clock, the physics module operates on a vsync clock, where the physics enters the next cycle at each negative edge of vsync signal. Since these two have very different periods, we needed a way for them to exchange data without any loss of information. We expected this to be a significant challenge, but the simple implementation of "pdone" and "newdata" successfully solved this problem. When checked on ModelSim, no data was being lost.

Whenever "pdone" is set by the physics module, the game logic module carries out most of its functionality. It first checks if the signal "goal" is set, in which case it checks whether the puck's last location was in player 1's side or player 2's side and increments one of their scores

accordingly. Further, the module sets the variable "stop_capture" to 1, so that our BRAM does not record any more locations until the game restarts. Then, it compares the current mallet location with the past mallet location from one "pdone" ago to calculate the mallet's velocity and sends all the necessary information, velocity and locations of all the objects, to the physics module, if the game is in playing state (states will be described in detail in a later paragraph). Finally, if "stop_capture" is 0, the module sets write enable to input a 60 bit long data, which is a concatenated version of all the objects' x and y location coordinates. It is important to note that the object recognition module sends the mallet locations to the game logic module every clock cycle, so many reported mallet locations between the two high's of "pdone" will be lost. We found out that this is not a problem since this implementation acts as a downsampler, blocking out some high frequency noise that may be present in the output of the object recognition module and smooth out the movement of the mallets in graphics.

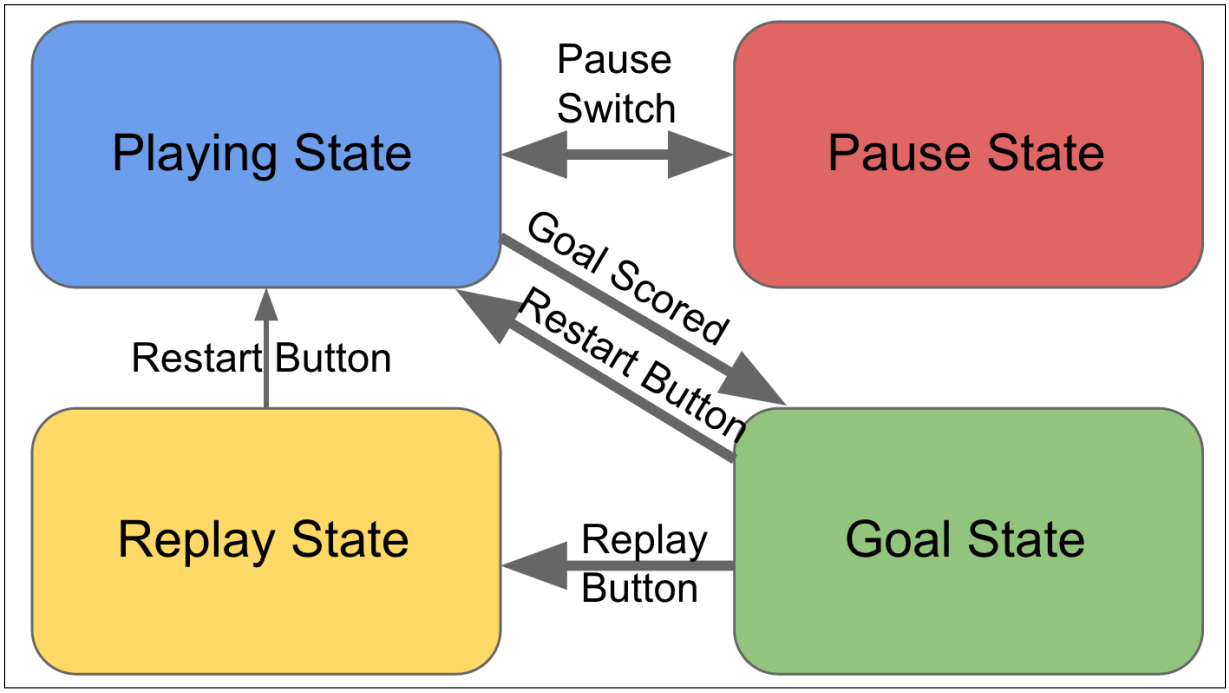

Figure 20: Game logic block diagram.

3.4.2.2 Communicating with the Graphics Module When "pdone" is set, the game logic module also sends the locations of the objects, two mallets and the puck, and the current score to the graphics module. The locations of the objects sent to the graphics module depends on the state of game logic module, which can be state 0 (playing state), state 1 (pause state), state 2 (goal state), and state 3 (replay state). Playing state is a self-explanatory one in which an air hockey game is in progress and the players are trying to score, so the module sends the new puck location as well as the current mallet location to the graphics module. In the pause state, the puck should remain at the same location until it goes back to playing state. Since "newdata" is not being set in this state, no calculation is done by the physics module to update the location of the puck. The same procedure is used in goal state, which the module enters when a goal is scored: mallets are free to move around while the puck is stuck in the goal. Lastly, when the module is in replay state, the output of the location storage BRAM is deconcatenated and fed to the graphics module. Since we are using a circular BRAM, the replay repeats itself in a constant loop until the module enters the playing state again. The state transition is shown in Figure 20. As shown in the figure, the module changes from one state to another either by a signal from the physics module or buttons and a switch that users

can trigger.

3.4.3 Implementation Process and Testing

The design and coding of this module was finished during the week after the Thanksgiving break. Since the module has to be tested on whether it communicates effectively with all the major modules, we could not test the module in isolation. We considered using ModelSim and wondered how plausible it would be to make a testbench for this module, but we decided that it would be a huge time sink that might not be worth the effort since we would have to create all the inputs for the module and whether the module is synchronized well with all major modules could not be determined until all the other modules were ready. So, we decided to test and debug (should there be problems) this during the integration after all the modules have been completed.

The integration did not occur until much later in the process than we had imagined and did not have much time to check whether this module was functioning well. No major modifications were done on this module after the initial design by the time we had the checkoff meeting. After the integration, we found out that all the objects were moving as expected, but the states were not functioning as expected. Whenever we turned on the pause switch, the graphics showed the pause state signs, but the puck was still moving. After the goal was scored, even when we were pressing the replay button, the objects in graphics were not moving as saved in the BRAM, which we checked was storing all the data correctly.

3.4.4 Review and Recommendations

If given more time, we would try to find out why the communication between the graphics module and the game logic module created some unexpected behaviors and debug to ensure that the movements of the objects reflect the state of the module.

One idea we could not carry out because of the deadline was potentially combining the game logic and physics module. Integrating two modules that are already lengthy may cause the module to become a lot more complex, but it would decrease the amount of communication to be done between the modules and shorten the critical path of data to be object recognition $\rightarrow$ physics $\rightarrow$ graphics, which would get rid of the need to implement many protective measures. Further this implementation will decrease the throughput between the object recognition and graphics module, which may be necessary should we continue to make more improvements to the game.

4 Integration: Review and Recommendations

As a team, we underestimated the amount of time required for integration. This can be attributed to three causes. First, we were generally overconfident in the correctness of our individual components, simply underestimating the number of mishandled edge cases. Second, some of our code was a bit messy, making it more difficult to catch simple bugs. With a compilation time of 40 minutes, every small mistake costed us. Third, we had made assumptions about the progress of other modules that were not expressed very clearly.

For example, the graphical interface is a key component for functional testing of our project as a whole. The physics and game logic components were relatively cumbersome to test independently, and for testing of those components we had assumed the completion of a functional graphical interface. Unfortunately, we had failed to express that expectation during development. By the time a functional graphics component was available, we had relatively little time to fix the issues that were exposed. Combined with the first two reasons above, this made the successful completion of our project difficult.

In retrospect, there a few ways in which this collaboration could have been carried out more effectively. First, the expectation of the progress of another module could have been expressed more explicitly. A clear expectation for a functional graphical interface for testing purposes might have influenced the graphical interface development to produce a basic version sooner. Second, upon realizing that a functional graphical interface would not be immediately available, it might have been possible to create more comprehensive independent tests using the hex display and VGA display.

Additionally, it would have been beneficial to attempt integration earlier on in the process. Instead of wiring together nearly finalized modules at the end, integrating simpler and less functional modules earlier in the pipeline would have been very helpful. It would have presented us with a better idea of the type and scope of integration issues earlier on in the timeline, giving us more flexibility in managing it.

Finally, we observed some issues during integration that we still do not understand. Although the three main components were functional in our own individual testing, there were issues when we combined all three. During the checkoff, we were able to present a successful integration of the graphics and object recognition components. With very minor and seemingly tangential changes, some later attempts produced functional objection recognition and physics engine, while others produced a functional physics engine with graphics. We were able to produce a version with all three components integrated for the video presentation.

Given the size of our project and the seemingly nondeterministic behavior of the integration, we suspect that we had violated some timing constraints during the integration process. Unfortunately, we had identified this problem relatively late, and we were unable to pinpoint the exact root cause.

5 Conclusion

Our finished product was a playable air hockey game with resets. The object recognition module found the sent the mallet locations through the game logic module to the physics module, which sent back the next puck location. The game logic module was then able to relay all the objects' locations to the graphics as well as the state of the game.

Despite the challenges, we enjoyed working on this project overall. We have gained a much deeper appreciation for the issues that arise when putting together large low-level projects. We would like to thank the entire 6.111 staff for all of their help. In particular, we would like to thank Luis for advising our project, Jose and Gim for investing many hours into the lab, and Michael Trice for providing valuable feedback on our proposal and presentation.

Another fun option to implement is the sound effect. We can save the collision sound and celebratory goal sound to a separate BRAM for the game logic to output whenever there is a collision or a goal. Since we already have the information about the velocity of the puck before and after the collision, we can make the loudness of the collision sound depend on the impulse of the puck so that collisions with higher velocity change in puck (harder collision) will result in a louder sound effect.

6 Code

All of our code is available at `https://github.com/axc/6111_air_hockey`.