6.111 Final Project Report

Two player Air Hockey game

Justin Graves
Matthew Guthmiller

Introduction

We attempted to implement a virtual two player air hockey game. This virtualization makes air hockey more accessible by eliminating the need for a dedicated room since air hockey tables are generally large and cumbersome to move; ours is comparatively quite portable. In our game the players are able to control the paddles by pressing buttons on the labkit and the game itself is rendered on the monitor. We chose to implement this air hockey game as our final project because the game itself presents a variety of technical challenges, including modeling the physics of the collisions of the puck with the paddles and walls to such a degree that the gameplay would render smooth graphics and look realistic. We also knew that making a game would allow for flexible goals since we could add various features once a basic game was implemented.

The basic concept of air hockey is that two players stand on opposite sides of a rectangular table and each control a paddle to strike a puck, which floats freely on a pocket of air to minimize friction. Each end of the table contains an opening that serves as a goal, and each player uses their paddle to defend their goal and attempt to score into their opponent's goal. The edges of the board are raised to allow the puck to bounce off the four sides of the table, and the surface of the table has small holes that blow air to allow the puck to glide freely.

Unfortunately there were some systemic issues with our approach that left some overlooked problems until too late, preventing us from completing the game. The biggest thing left unfinished was the physics modeling behind the game. Primarily we waited too long to integrate our work since we worked in parallel and once we discovered the physics was not working we both halted progress on all other modules and worked concurrently on getting the physics to work.

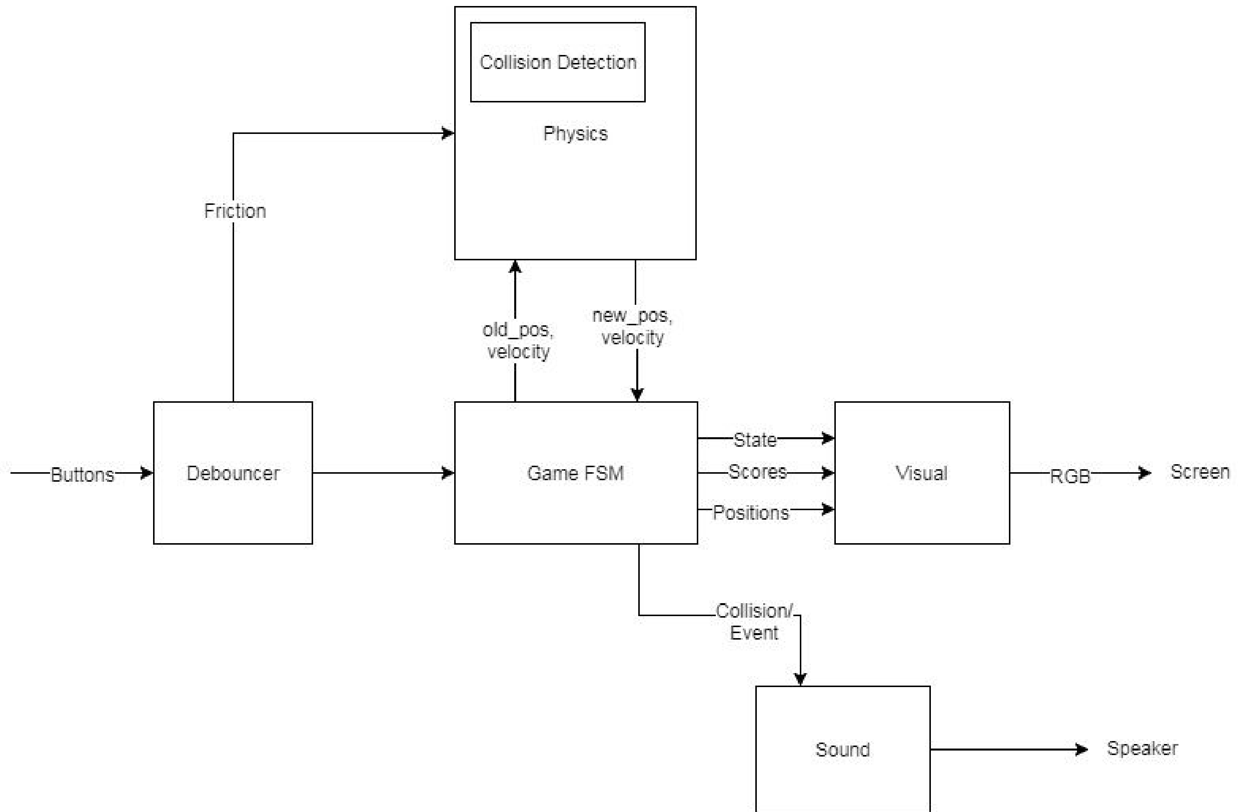
As a result, during the checkoff we were only able to show a rudimentary version of the game that contained the visuals of the board, paddles, and pucks. While the paddles did move, because the physics was not working the puck was not able to move and therefore you could not play the game.

Summary

Before beginning the implementation we broke the game down into three main modules that would allow us to effectively work in parallel so that progress would not be bottlenecked as we waited for each other to finish building a component. The three main modules were a module to control the physics for the game, a module to control the game logic of the game, and a module to control all the graphics for the game.

Top Level View Block Diagram

The physics module was tasked with taking the positions of the paddles, board geometry, and positions of the puck as inputs and returning the position of the puck, updating its velocity in response to collisions and friction. The game logic module was tasked with connecting the graphics module to the physics module. The game logic module would also keep track of scores and other game state information. The graphics module was tasked with producing everything that would be displayed on the screen including the puck and paddles in addition to any gameplay information such as score and who won the game. Matthew was responsible for getting the physics module up and running, and Justin was responsible for the game logic and the graphics.



Block Diagram for final Air Hockey game Design

Modules

collision_checker (Matt)

This module takes the positions of the puck and each paddle and outputs whether a collision has occurred and of what type. It has 7 inputs -- clock; 11-bit inputs for the x positions of each of the puck, paddle1, and paddle2; and 10-bit inputs for each of their y positions. It has one 3-bit registered output to specify the type of collision detected (including a value for no collision). The module is also parametrized to allow the dimensions of the board, boundaries of the goals, paddle radius, and puck radius to be changed from default values.

At the positive edge of the clock, the module calculates the squared distance between the puck and each paddle to compare to the square of the sum of the puck and paddle radii in order to determine whether they overlap. The squares of these values was chosen to avoid having to calculate square root to determine the actual distance between the puck and paddles. In addition, the distance calculations perform subtraction in different orders depending on the relative values in order to avoid signed arithmetic since we only care about the absolute value anyway.

The module then checks each possible type of collision the puck could encounter -- with paddle1, paddle2, the top edge of the board, the bottom edge of the board, the left edge of the

board outside the goal, the right edge of the board outside the goal, or the goal itself (if the puck is in contact with the left or right edge and none of the above conditions apply).

A simple test fixture was created to place the puck and paddles in different positions to test that the output collision type in each case was as expected. This proved very useful in resolving a few bugs, as well as eventually confirming the module works as expected.

physics (Matt)

This module updates the position of the puck, modeling collisions and friction. It has 11-bit inputs for the x coordinate of the puck, paddle1, and paddle2; 10-bit inputs for the y coordinate of each; a 4-bit input to set the level of friction; a 3-bit input for the collision_type detected by the collision detector; and an input for each of clock and vsync. It uses 11-bit and 10-bit registered outputs for the updated x and y coordinates of the puck, respectively.

The module copies the the positions of the puck and paddles into the lowest order bits of signed registers (which are one bit wider) in order to take advantage of signed arithmetic in computing the puck velocity and position updates. The puck position is initialized, and then puck position and velocity state is maintained internally since this module is the only thing modifying those values. A final version would incorporate a reset input in order to reset the puck velocities to zero and allow the puck position to be re-initialized, such as when starting a new game.

The module performs updates on the negative edge of vsync, although it was planned to transition this to updates on the positive edge of the system clock (while vcount is beyond the visible lines of the screen). The latest paddle positions are observed, and the x and y components of paddle velocities updated as the difference between their new and previous locations. If a collision has not occurred, the puck velocity is decreased in both dimensions by the set amount of friction once every 30 updates (approximately once per second). If the collision input indicates a collision with one of the walls, the appropriate component of the puck's velocity is negated. If a paddle collision has been detected, the point of collision is calculated (i.e. where the puck and paddle overlap) and a vector from the paddle to that point constructed. The relative velocity of the puck and the paddle in that direction is added to the velocity of the puck (per the math below). Denominators are approximated by powers of two for now in an attempt to get something working before possibly performing actual division for the utmost in accuracy later on in development.

$$P_{col} = \frac{C_{puck} \cdot r_{puck} + C_{pad} \cdot r_{pad}}{r_{puck} + r_{pad}}$$

$\vec{V}, \theta = atan($

$\theta' = -\theta_v - 2\theta_d$

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

$$\vec{V}' = \vec{V} - 2(\hat{n}_x V_x + \hat{n}_y V_y)\hat{n}$$

$$V_x' = V_x - 2(\hat{n}_x V_x + \hat{n}_y V_y)\hat{n}_x$$

$$V_x' = V_x - 2\left(\frac{n_x^2 V_x}{n_x^2 + n_y^2} + \frac{n_x n_y V_y}{n_x^2 + n_y^2}\right)$$

$$V_y' = V_y - 2\left(\frac{n_x n_y V_x}{n_x^2 + n_y^2} + \frac{n_y^2 V_y}{n_x^2 + n_y^2}\right)$$

Collision component

paddle movement component

$$\vec{V}' = \vec{V} - (\hat{n} \cdot \vec{V}_{pad})\hat{n}$$

Should V_x, V_y be $(puck_{vx} - paddle_{vx})$?

$$P_{col\,x} = \frac{puck_x \cdot r_{puck} + pad_x \cdot r_{pad}}{r_{puck} + r_{pad}}$$

$n_x = (P_{col\,x} - pad_x)$

$n_y = (P_{col\,y} - pad_y)$

$\hat{n}_x = n_x / \sqrt{n_x^2 + n_y^2}$

$\hat{n}_y = \frac{n_y}{\sqrt{n_x^2 + n_y^2}}$

$\hat{n}_x \hat{n}_y = \frac{n_x n_y}{n_x^2 + n_y^2}$

$\hat{n}_x \hat{n}_x = \frac{n_x^2}{n_x^2 + n_y^2}$

$\hat{n}_y^2 = \frac{n_y^2}{n_x^2 + n_y^2}$

Initially we believed this to be straightforward enough to test primarily just by integrating with the graphics and observing puck and paddle motion on the screen. This is partly due to the combination of the fact that ISE synthesized the module without any errors and the realization that any working test would only reveal whether the math was being calculated correctly, which we viewed as the easier problem than actually using math that properly modeled the desired gameplay. Unfortunately, however, integration of this module with the graphics did not take place until several weeks later, delaying the realization that this module was not working properly until after Thanksgiving. A simple test fixture was constructed to observe the module's behavior and assist in debugging. However after a cumulative 20 or so hours of collaborative effort to debug this, we were unsuccessful.

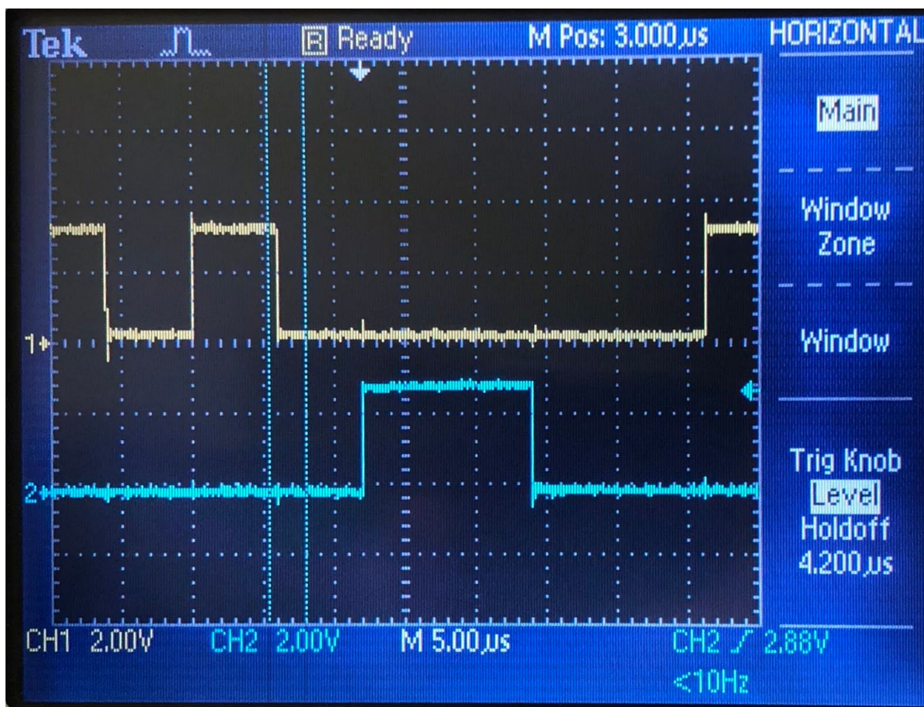
nes (Matt)

This module generates the necessary control signals for the NES controller and collects returned data, output as the status of each individual button. Its inputs are clock and a data wire

for the serial output from the NES controller; its outputs are the latch and pulse wires to the controller and an 8-bit registered output with the most recent state of each of the 8 controller buttons.

The interface with the controller was designed based on the controller specifications found at https://tresi.github.io/nes/. A counter is used to generate a clock signal with a roughly 12 microsecond period; a simple finite state machine uses a 4-bit counter to transition between 3 states to deliver the appropriate signals to the NES controller, while collecting each bit of the serial signal on successive cycles of the output pulse signal.

A test module was used to debug and test the module by simulating the NES controller's response throughout the polling cycle. While everything worked perfectly in simulation, and the appropriate control signals were observed on the oscilloscope, extensive debugging efforts involving multiple TAs, as well as Joe, were unsuccessful in actually getting the controller to output any data signal. In total at least 8 hours were spent just trying unsuccessfully to debug this (after initially writing the module).



puck (Matt)

This module is responsible for generating the pixels that make up our circular objects, including the puck and paddles. It has 5 inputs -- clock, an 11-bit x coordinate for the center of the circle, 11 bits for the current hcount signal, and 10 bits each for the y coordinate and vcount. It outputs a 24-bit registered pixel, representing red, blue, and green with 8 bits each. Three parameters

can be used to configure the radius of the circle, the color of the circle, and the default non-circle pixel color.

Like the collision detector, the module compares the square of the distance between (hcount, vcount) and (x,y) to the square of the circle's radius. If (hcount, vcount) is within the circle, the output pixel contains the color of the circle; otherwise it is set to the default pixel color (in our case white since that is the color of the board). This construction was based on the course notes about drawing circles.

This module was easily tested and debugged by simply using it and observing the output on the screen.

visual (Justin)

The visual module was a standalone module that took inputs of the location of the three main game objects, the puck and two paddles. The visual module was also responsible for displaying anything related to text to display score, timer, or any other game information. The submodule xvga was used to interface with VGA and during each clock cycle the visual module was responsible to output a color to be assigned to a pixel given a location. SInce we were using a display of 1024x768 pixels we needed a refresh rate of 60Hz so we used the 65 MHz system clock with the goal to generate approximately one pixel per clock cycle.

The proposed plan for the visual model was to first draw the board and have the puck and paddles overlay it on the board. The idea behind this was that the puck and a paddles would be defined by geometry parameter and use the puck module to render them. The puck module would generate a colored pixel that was in the specified geometry bounds and if it was outside the bounds generate white pixel so that it would not conflict with the board's background. The next phase was to generate the text on the screen and to add visual aesthetics to the existing board. The latter was never fully implemented due to the mis allotment of time.

The basic idea for the text module was to read text information for memory and output the pixels accordingly. We intended to use a tile map scheme where we treated each ASCII character as a tile. The seven bit ASCII code for each character represented a tile that contained its specific pattern in a bitmap scheme. The font scheme was going to be a 8x16 pixel pattern and stored in ROM. The font ROM was implemented but the text generation was not fully development due to mis allotment of time.

To implement the visual module we used the existing framework from the pong game to interface with VGA. The visual model was definitely the most important module for debugging all other modules and itself. One mistake that we made of the implementation of the air hockey game was we neglected to hook up the modules from the physics area to check if there were working properly early enough.

GameFSM (Justin)

The idea behind the GameFSM module was that it would keep track of the state of the game and communicate with the physics module and the graphics module. The plan was to have this module feed the current positions to the graphics module so that objects could be rendered on the screen and to send the current positions to the physics module to determine if positions were going to change due to events like collisions in the game. The physics module would update the current positions in this module and this module would then in turn relay that information to the graphic module. This module was also responsible for provided a user interface to the players by controlling a start screen menu, keeping track of game state parameters like score, and pause screen. It was estimated that this module would have taken less than 10 hours to implement which why it was chosen to be done after Thanksgiving break. Also since this module was communicating with these other two modules it would be harder to test this module in isolation.

This module was never fully implemented only a framework was done. Part of the reason it was not fully implemented was because a lot of time that was dedicated to working on this module after Thanksgiving break was spent on debugging and re-writing the physics module since it failed to work properly once integration started. The idea behind waiting to implement this module after the break was that we wanted the visuals to be hooked up with the physics engine fully working so that we could debug the game and make sure the important game actions actually updated in the state machine. WIth hindsight we found that this was an ineffective way and it wasted time as it bottlenecked the design of the game. We realize that it would have been better to have implemented the GameFSM module completely without the physics module working and instead used buttons and switches to update the game state to debug the module.

If we did this project over again this module should have been implemented alongside the graphics module and the game stated could have just been controlled by switches (number in binary representing each state) and buttons or switches to control scoring. This would have allowed us to see if this module updated the graphics module correctly and to ensure this module was communicating correctly to the graphics module. Also this module was to serve a the communicator to the sound module, that was never implemented, to alert the sound module the proper time and type of sound to play based on the current state and event within the game.

Integration

Our biggest downfall with the implementation of the air hockey game was the lack of testing and integrating of the physics module with the graphics module. We underestimated how much time

the integration was going to take because we assumed that each module would be at least partially working before Thanksgiving break and integration would make any lingering issues apparent and easy to fix. In reality that proved not to be the case, and to make things worse the integration did not occur until after Thanksgiving break and was a huge delay for our team in getting the basics of the game up and running. With hindsight, we realize that we should have made a separate trivial physics module that provided basic predictable motion for the puck just for testing the rest of the game modules.

Conclusion

In short, we should have tested and sought help sooner. There were also a few communication issues regarding coordinate systems that caused some minor confusion, so we should have perhaps spelled out our specs just a little bit more completely from the onset. Our biggest recommendation to future teams building something like this is to get the most basic version of the game running absolutely as soon as possible.

While we were not successful in completing the game as hoped, we did end up with a lot of things that worked well, and we spent a tremendous amount of time attempting to fix things that nearly worked, such as the NES controller interface and the physics module. We had a solid design for each major components, but in the case of things like the NES and physics modules persistently encountered subtle errors that rendered them completely useless; it's unclear what exactly was wrong with either, but had those last small errors been resolved, we would have had a fully functional game. And if they hadn't proven so difficult to debug, we would have made much more progress on further features.