FencePGA
Preliminary Report

Gaurab Das
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
gaurabd@mit.edu

Muhender Raj Rajvee
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
muhender@mit.edu

***Abstract*—We implement an interactive multiplayer 2-player fencing-styled game, giving players the ability to attack and defend using hand gestures and body motions. Player positions and arm actions are tracked in real time and manifested within game logic. The game ends when one player loses all their health because of attacking swipes/stabs by the opponent. This project utilizes HDMI graphics, image cameras and motion tracking, networking, and game design methodologies.**

***Index Terms*—Computer Vision, Networking, Collision Detection, Graphics**

I. Introduction

Motion sensing has been a common theme in video game design, such as in the Nintendo Wii or the Xbox Kinect, enabling players to use their entire bodies to control the game, and allowing for a more immersive gaming experience. A Field Programmable Gate Array (FPGA) is ideal for designing these kinds of games because of its ability to execute in parallel a wide range of tasks from vision sensing and detection to networking and graphical interfaces. Inspired by our desire to create a similar gaming experience with friends and family, we leverage FPGAs to construct a fully-functional interactive multiplayer fencing game. A high level overview of the entire system is depicted in Fig. 1, and the hardware game setup is depicted in Fig. 2. The Camera detects the motions of

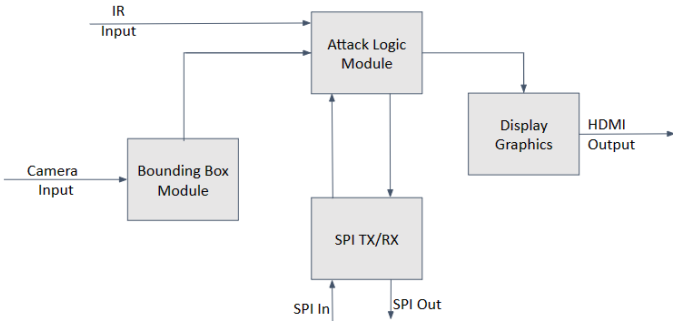

Fig. 1. Block Diagram of system consisting of higher-level modules

the players and their remotes, and the IR sensor detects what action the user wants to take. This paper is structured as follows: Section II discusses the bounding box module in-depth, Section III deals with the Attack Logic Module which also encompasses the SPI TX/RX interface, Section IV discusses the interactive display interface, Section V evaluates the performance of our project, and Section VI summarizes our achievements, goals and lessons learned. The link to our public repository is github.com/gaurab1/6205-FencePGA.

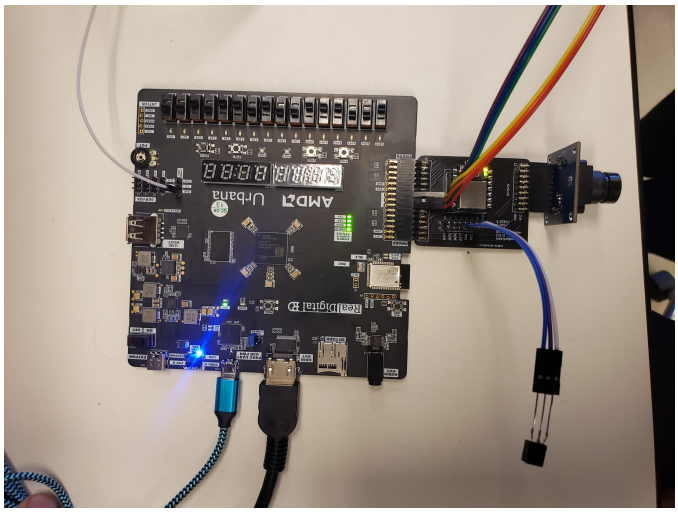

Fig. 2. Hardware Setup including FPGA, HDMI cable, IR sensor and SPI wires

II. Bounding Box Module (Gaurab)

The task of the bounding box module is two-fold: one is to track the position and size of the player body, and the other is to track the position of the hand that holds the saber. The motivation for detecting the player's body and size is to provide a way for the player to move around the screen and change the size of their body to avoid getting hit by their opponent. This is a hard problem because we want to do accurate real-time tracking of the objects with almost no memory overhead and with insignificant delay. We achieve these objectives by utilizing a bounding box algorithm and making a few reasonable assumptions. The player must tape a sizeable piece of paper on the remote of a particular predetermined color, and the player must also wear a shirt/top of a color different from the color on the remote. The bounding box module aims to track these two different colored objects separately and with low latency, and give accurate descriptions of dimensions and size of these objects. Refer to Fig. 4 for a result of the completed implementation of the algorithm.

A. Algorithm

The high-level flow for the algorithm is shown in Fig. 3. The algorithm runs in two stages. The first stage is to find the center

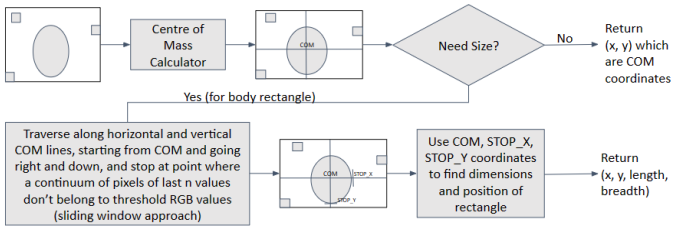

Fig. 3. High-level overview of bounding box algorithm

of mass of the pixels that fall within the color range threshold. Once we find the coordinates, we scan the frame image along the horizontal and vertical lines containing the center of mass to see how far we could go along these lines to form a rectangle that can contain the object, with a flexible parameter of error threshold. This algorithm is a near-robust estimator of the true bounding box, as outliers like noise present outside the object do not greatly affect the rectangle. Based on this approach, the current frame will include the height and width estimates of the rectangle using the center of mass estimates of the rectangle from the previous frame, but because the camera produces frames at ∼30fps while the system works at ∼60fps, this method should be frame-consistent.

This novel algorithm leverages the insight that the scanning of pixels on the monitor happens in a very systematic manner, and this inherent ordering of the pixels makes it efficient and fast to utilize center of mass information to predict the bounds of the tracked object.

B. Implementation Details

We implemented the bounding box algorithm described previously and tested it extensively via testbenches and FPGA builds. We also extensively searched for the right threshold parameters for our bounding box algorithm. As observed in Fig. 4, our bounding box module works well with an object of a specific color, and even when we shift the object or modify the shape by tilting it, the bounding box adapts well to fit the object without any considerable delay. We are able to track multiple objects using the bounding box, and thus this algorithm meets the specifications of our game. Previously, we only used one channel out of the available 6 channels (RGB and YCrCb). An additional insight that we had was to utilize more than one channel for masking, which allowed us to be more precise with object tracking by filtering out pixels that did not belong to the tracked object.

III. Attack Logic Module (Muhender)

A. Serial Peripheral Interface

We use the Serial Peripheral Interface (SPI) protocol to enable communication between the FPGAs.

As illustrated in Fig. 5 by the Locations Transmitter and Syncer modules, the Locations Transmitter transmits the

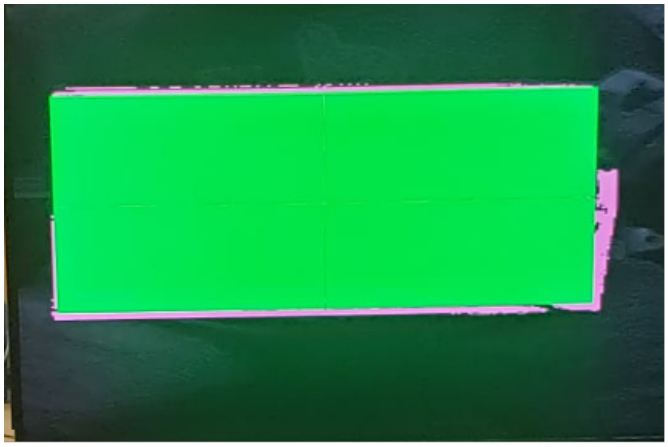

Fig. 4. Bounding Box algorithm generating a green rectangle encompassing the pink plastic object

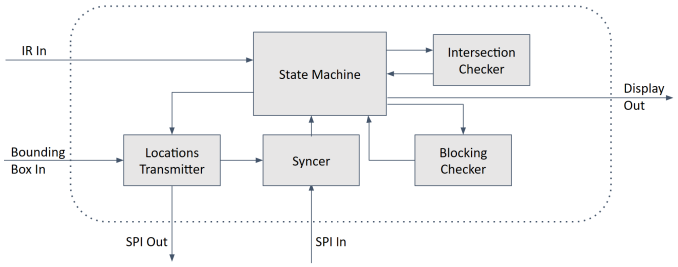

Fig. 5. The attack logic module

player's current state and the player score status to the opponent through a SPI transmitter. The syncer waits for input from both the Locations Transmitter and the SPI receiver before sending both information to the State Machine for processing.

The SPI data is organized as a stream of bits as in Fig. 6. Each FPGA sends its current state and whether the player scored a point in the previous cycle to the other FPGA before performing any game calculations. This way, both the FPGAs are synced at the start of every video frame's game logic computation period. Each player starts with 5 health points which requires 3 bits, each position requires 11 + 10 = 21 bits, and the saber can be either resting, lunging, or blocking, which requires 2 bits to store the state. In total, the data is 89 bits long. Adding the extra bit to indicate whether the player scored brings this total to 90 bits.

On initial flashing, the FPGAs use one of the SPI wires to send a constant LOW to each other. When the game is started, the first FPGA to start sends a constant HI to the other FPGA. The other FPGA receives this, sends a constant HI back to the first FPGA, and transitions to the game. The first FPGA receives this too and similarly transitions to the game.

B. State Machine

The core game mechanics are implemented as a finite state machine (FSM). This FSM module takes in the SPI output from the other player, the IR input from the IR saber, and the bounding box output as inputs.

Health	Saber position	Rectangle position	Saber State	Saber Starting Position

Fig. 6. The player data

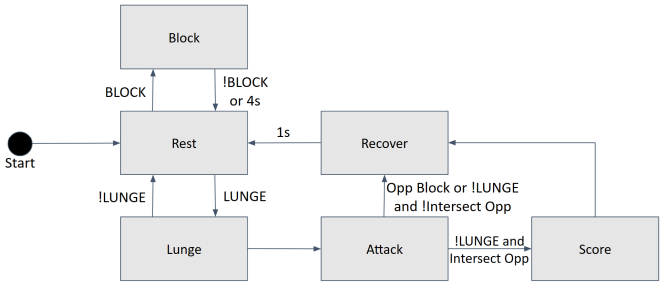

Fig. 7. The finite state machine

The FSM starts at *Rest*. On receiving the ”BLOCK” signal from the IR saber, it transitions to *Block*. It transitions back to *Rest* when the ”BLOCK” signal is no longer being received or in 4 seconds, whichever is shorter. In *Block*, the player can block an opponent's active attack, which will be described later.

The FSM transitions to *Lunge* on receiving the ”LUNGE” signal from the IR saber, and transitions back to *Rest* when it is no longer being received. The current saber position is stored on transition to *Lunge*. While in *Lunge* and *Attack*, a line is actively drawn from this saved position to the saber location in the current frame.

The FSM transitions to *Attack* immediately from *Lunge*. There are multiple possibilities from this state.

1) If the ”LUNGE” signal is no longer being received, and the line from the saber start position to the current saber location intersects the bounding box of the opponent, then the FSM transitions to *Score*. The intersection algorithm is described in a later subsection.
2) If the ”LUNGE” signal is no longer being received but the line described in the previous case does not intersect the opponent's bounding box, then the FSM transitions to *Recover*.
3) If the saber position collides with the opponent's saber position while the opponent is in *Block*, then the FSM transitions to *Recover*.

In *Score*, the opponent's health is decremented by 1. The FSM transitions to *Recover* in the next clock cycle.

The FSM stays in *Recover* for 1 second before transitioning back to *Rest*. The player can not take any action while in *Recover*. This acts as a cooldown period to prevent blind slashing without a strategy.

C. Saber Collision Detector

This module checks if two sabers are within a given collision distance to each other using the Manhattan distance between the two sabers. This module is only used when the player's FSM is in *Attack* and the opponent's FSM is in *Block*.

D. Intersection Detector

This module calculates if a line segment intersects a rectangle. The FSM uses this module to determine if a transition to *Score* should be performed. This module is only used when the player's FSM is in *Attack* and the ”LUNGE” signal is no longer being received.

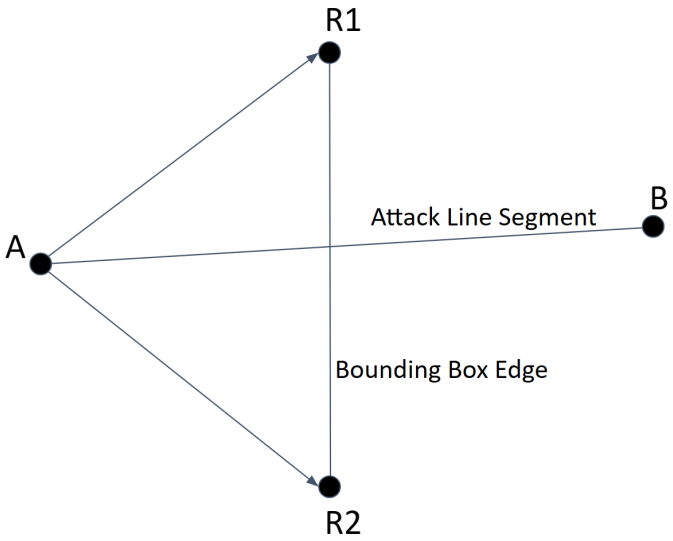

Fig. 8. Example of a case for the intersection logic

In order to do this, the module computes intersections of the line segment with each of the 4 line segments forming the rectangle. In order to make this efficient, it exploits the symmetry of the sides of the rectangle: two sides are vertical and two sides are horizontal.

For the vertical case illustrated in Fig. 8, the module first checks if the x coordinate of the rectangle vertices are in between those of the two ends of the line segment. If not, then the line segments can not intersect.

If the above check passes, then the module computes vectors AR1, AB, and AR2. If the cross product AR1 $\times$ AB and the cross product AR2 $\times$ AB have opposite signs, then the line segment AB intersects line segment R1R2. The module follows a similar logic to compute the intersection with the horizontal sides.

After these intersections are computed for all 4 sides, the module returns true if the line segment intersects at least 2 sides of the rectangle and false otherwise.

IV. Display Module (Gaurab)

Using all of the outputs from the top-level attack logic module, we implement a simplistic game interface that gives complete information about what is happening in the game. As seen in Fig. 9, we display the following information through HDMI:

- The health bars of both players, which are calculated in the attack module based on how players interact.
- The size and positions of the rectangles of both the players which are captured by the camera (for the player)

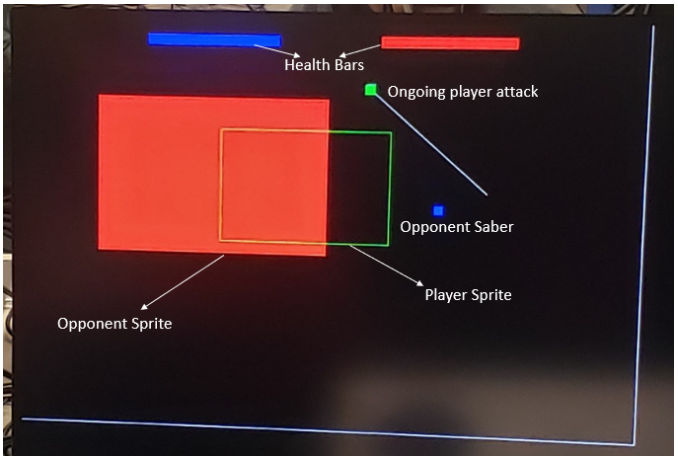

Fig. 9. Game Frontend Display

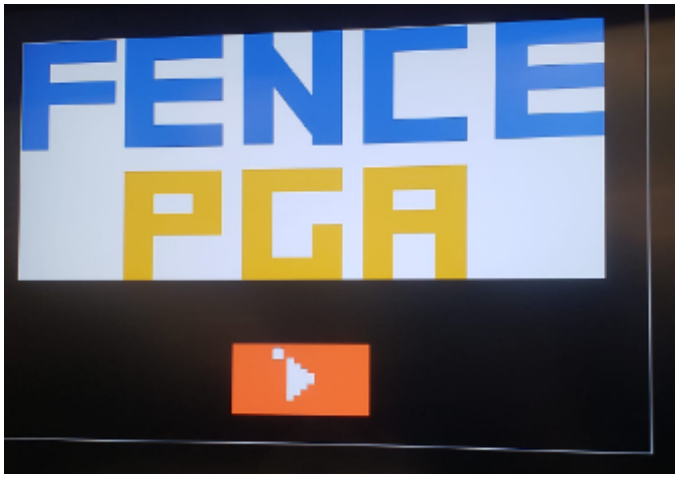

Fig. 10. Game Start Screen

or by the SPI (for opponent). Because both rectangles need to be visible, the player is represented as a rectangle with only an outline and no fill color.

- The positions and status of the saber points of both the player and opponent on the screen. The color of the saber depends on the state the saber is in. Additionally, to differentiate between the player and opponent saber, the player saber is highlighted and has a cool tracing effect described in detail in IV-B.
- When the player does attack, they start at an "anchor point", and then finish their attack at another point. The line joining the two points determines the attack (as described in Section III), and so we create a module inspired by Bresenham's line-drawing algorithm [1] described in more detail in IV-A that draws a line efficiently between the anchor point and the current point of the saber.

Apart from the main game display, we also created visually appealing opening and ending pages that allow players to transition in and out of the game synchronously. The start screen is as seen in Fig. 10, where the player presses "OK" to transition into the game. We also show an end screen when any player's health goes to 0. This screen is personalized to show who won and lost the game. In the process of implementing front-end display modules, we also created useful shape libraries including lines, triangles and rectangular outlines.

A. Line Drawing Algorithm

We implemented a line drawing algorithm that is efficient (uses integer addition and subtraction only) and fits in well with the raster video pipeline of generating pixels from the top-left to the bottom-right corner. We initially built a naive line generator based on integer multiplication and comparisons to match slopes, but we observed that it didn't work as well because the generated line wasn't continuous. To fix this, we researched line drawing algorithms and implemented a variation of Bresenham's line drawing algorithm (seen in Fig. 11).

The Bresenham's algorithm resembles a Pulse Density Modulator in that it keeps track of a quantization error and lays down lines based on whether it is negative or positive. Say for example that we want to draw a line with slope between 0 and 1 from (x_0, y_0) to (x_1, y_1), with $x_0 < x_1$. This algorithm starts at the initial point (x_0, y_0), and in each iteration it increases x_0 by 1 and increases y_0 by 1 or 0 depending on what the state is, and it continues this iteration until we reach (x_1, y_1). The error is only dependent on a linear combination of the endpoints and is updated each iteration. Notablly, it doesn't involve any expensive variable multiplication and division operations.

We extend this case to allow for a wide range in slopes by making minor modifications to the original algorithm. We adapted this algorithm to fit the raster pattern for video generation and achieved a line generator that produces a line every frame given the endpoints with very simple operations and no memory overhead.

Fig. 11. Screen displaying line from anchor point to saber point, with trace effects also shown

B. Displaying Saber Trace

To simulate the visuals of a saber's trace, we created a module that stores the past history of the saber positions in a manner very similar to pipelining. We then use this history of information to create a trail of past saber positions. More specifically, we use alternating frames in the history of the 12 past saber positions and decrease the size and color intensity of the saber depending on how far back it was in the history. This imparts an animated effect to the pixels until they completely dim out. The result of the trace display module can be seen in Fig. 11.

V. Evaluation of Project

A. Latency and Throughput

At a high level, the game experience is uninterrupted without any significant delays. In order to achieve this, we needed to fit all game logic computations between two frames, which is a lot of time. Because of this relatively loose requirement, most of the modules just rely on a ready/valid protocol to pass results to each other, with the valid bit set in the clock cycle the computation succeeded. The bounding box module produces its output once per frame. Hence, the attack module also completes one execution cycle per frame.

The SPI synchronization process needs to happen between two frames. The FPGAs send and receive 90 bits of information simultaneously every frame, and the SPI transmitter uses a data clock period of 100 pixel clock cycles. This leads to a total transmission period of 9000 pixel clock cycles, which is comfortably within the 1.2375 million pixels per frame.

B. Timing

We split up large combinational computations to fit between two rising clock edges wherever possible. Since each multiplication in the cross-product computation can happen in parallel with each other, the entire computation fits in one clock cycle and can be done combinationally. However, this still remains the critical path.

The project has a positive post-synthesis slack of 1.115 ns.

C. BRAMs and DSPs

Storing the camera output in a frame buffer takes up most of the available BRAMs on the FPGA. Because of this, we needed to budget the remaining BRAMs to store the opening screen image by using a very small palette of 4 colors. We use a total of 68 out of 75 available BRAMs.

We use 26 out of 120 available DSPs. Most of the DSP usage is for the multiplications in computing the vector cross products in the Intersection Detector module.

VI. Project Goals and Reflections

A. Individual Contributions

At a higher level, Gaurab worked on the Bounding box and display modules while Muhender worked on the attack logic module and all relevant submodules including the SPI communication.

More specifically, some of Gaurab's contributions to the project include the conception and implementations of the bounding box module, processed IR input module, and all front-end display modules such as the start screen display, end screen logic to display the winner and loser, and the in-game displays including the players, sabers, health bars and attacking lines. Some of the things that Gaurab is especially proud of is the conception of a novel bounding box algorithm, and his research in efficient line drawing algorithms.

Muhender's contributions include clearly defining the game mechanics and implementing the core game loop including the synchronization step, the game logic FSM, and the collision and intersection detectors. Of particular interest is the implementation of the intersection module with cross products that fits perfectly within two successive rising clock edges.

Both Muhender and Gaurab were together pivotal in brainstorming ideas relating to the overall game and individual modules and the integration of individual modules to form the complete game.

B. Project Features and Checkoff List

Based on our discussions during the presentation and final stretch meeting, we have a list of completed items from the checkoff list.

1) We successfully achieved the base commitment of creating and integrating the Bounding Box, Attack Logic and Display Graphics modules to have a 2-player playable game.
2) We also achieve the goals in the checkoff list. We inplement health bars that are functional and updated properly when an attack occurs. We have also added functionality for interactive start and end game scenes. In the end scene, the player is informed of whether they won or lost the game.
3) We we able to achieve some of the stretch goals from the checkoff list. In particular, we implemented the saber tracing effect.

Acknowledgements

We would like to thank the staff of 6.205, including professor Joseph Steinmeyer, the LAs, and the TAs of the class for providing us with great constructive criticism and for helping us debug some of our code.

References

[1] Zingl, Alois. "A Rasterizing Algorithm for Drawing Curves." (2012).

Appendix

The players need to set up before starting the game.

1) In order for the FPGA cameras to detect the bounding boxes of the players, tape a pink envelope to the body.
2) Stand at a distance of 1-2 meters from the camera.
3) Attach a white piece of paper to the back of the remote in order for the FPGA to track it.
4) Ensure proper white lighting and a dark background.