3D PONG!

Zitong Chen
EECS Department
MIT
Cambridge, MA
zitongc@mit.edu

Eugene Lee
EECS Department
MIT
Cambridge, MA
travisb@mit.edu

***Abstract*—We present a design for a 3D PONG game implemented on FPGA and utilizing HDMI monitors alongside a digital joystick and camera controller. This hardware utilizes the SPI communication protocol to share information between the two players and a simple switch to toggle between the control mechanisms. We implement the 3D visual rendering of the game through ray casting and the 3D logic with a module that calculates collision physics and updates the game state. We did not get to integrate the game logic modules with the 3D rendering logic. The final output of each module are discussed separately in Section IV. Control Mechanism and Section VI. 3D Display. We evaluate the game quality through smoothness of gameplay, hardware costs, and casework evaluation of game behavior both in simulation and hardware. We evaluate the performance and quality of the 3D pipeline design through the frame rate, latency, and cleanness of the 3D rendering.**

I. Introduction

Welcome to 3D PONG! In this project, we aim to elevate the classic Pong game with a 3D twist implemented via 2 interconnected FPGAs. In 3D PONG!, you get to play Pong with a friend from a first-person perspective. 3D PONG has two modes of control, allowing you to control your paddle using either the joystick or simply moving your hand in front of a camera. With 3D PONG!, we aim to provide you with high-quality 3D rendering on a 2D display and a smooth, seamless gaming experience. With other key features like a real-time scoreboard and a camera movement controller, we believe you will have a wonderful time with 3D PONG!.

II. Physical Construction

The projector itself consists of:

- 2 Xilinx Spartan-7 XC7S50-CSGA324 FPGAs
- 2 digital joysticks
- 2 cameras
- 2 monitors
- 2 HDMI cables
- wires for SPI communication

The 2 FPGA game controllers are identical in their physical construction. To distinguish the 2 controllers, we call Player 1's controller "Controller 1", and Player 2's controller "Controller 2". Controller 1 will serve as the main controller that initiates SPI communication between the 2 FPGAs (details in the SPI Communication section).

Special thanks to Professor Joe, Darren, and Ivy, without whom this project would be unthinkable.

III. Block Diagrams

A. System Block Diagram

As shown in Fig. 1, the player interacts with two different types of paddle controls: the joystick controller and the camera controller. Additionally, the player can set the x and y positions of the camera from their perspective. The coordinates of the paddle and camera are initialized to the center of the frame; these values will not be 0, since all coordinates' values are positive. The controllers send the x and y coordinates values of the paddle to the control selector, which then chooses one set of values based on a single switch. The coordinates of the paddle are then passed to the collision physics calculator, which can then update the game state appropriately. The game state controller contains information about the positions of the objects(paddle, ball) and the scores of the players, which it can then pass to the 3D Ray Caster and the renderer for the ball, paddle, and scoreboard. In parallel with the system, there is a camera position controller where the player can set the camera perspective of their monitors on a smaller subset of the entire frame. The information about the player camera coordinates is relevant for all 3D objects in the game, so these sets of coordinates are also passed to the appropriate modules to produce the final visual output.

B. Block Diagram of Display Modules

Unfortunately, we were only able to deliver part of the display pipeline in our original design. A detailed breakdown of the original display modules at a sub-module level is shown in Fig. 2. The Video Signal Generator iterates through a frame pixel by pixel, prompting the 3D Ray Caster module to render the background environment (floor, ceiling, and walls), which is stored in a frame buffer. The Ball Renderer renders the pong ball on top of the background. To ensure the game can run smoothly, we decided to use a preloaded image sprite for the pong ball. Instead of computing the lighting and shading of the ball in each frame, we will calculate the projection of the virtual 3D ball on our 2D game screen using the Perspective Projector. The Image Scaler then rescales the ball image to its corresponding size, which will be added to the final display in Video Mux. The Scoreboard Renderer renders the current player scores on screen using an image sprite lookup mechanism. The Base 10 Converter converts the base 2 scores stored in game states to base 10 using the dabble double

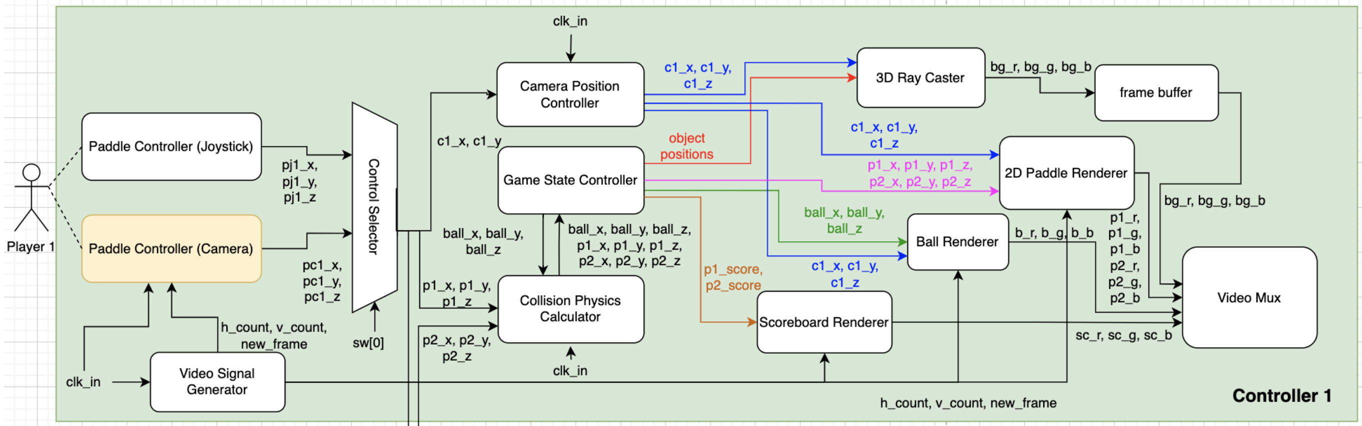

Fig. 1. Block Diagram for Controller 1

algorithm, and Sprite Look UP finds the corresponding letters and numbers on a preloaded sprite sheet. Similar to the pong ball, the scoreboard is also added to the final display in Video Mux. The Paddle Renderer shares a similar structure with the Ball Renderer to render 2 paddles on screen.

In our final implementation, the 3D pipeline functions as shown in Fig. 3. Using the camera position and camera direction information, the Ray Caster computes the new scene on each new_frame signal from the Video Signal Generator. The scene information is stored in the Frame Buffer, and the Background Painter reads from the Frame Buffer to render the RGB color for each pixel on the screen. The detailed design of the Ray Caster is discussed in *Section VI. B. Second Attempt: Raycasting*.

IV. Control Mechanism

A. Joystick Control

We use a digital joystick control corresponding to one of the values of a single switch on the FPGA. This is functionally identical to the movement of image sprites on a 2D plane corresponding to four directions controlled by four switches. Based on the state of the joystick, the control module outputs two values, $pj1_x$ and $pj1_y$, to the control selector(x being 10 bits and y being 9). These values correspond to the vertical and horizontal coordinates of the player's paddle. Currently, we allow for diagonal movement of the paddles, but fix the paddles in the z-direction.

B. Camera Control

We use a module for controlling the camera and refactoring the camera output similarly as used in the lab previously. We then have a dual-port BRAM functioning as a frame buffer, where the camera writes data on one side and the rendering stream pulls out the color channel data from the other. For the camera-controlled paddle, we have a standardized paddle object that we use to threshold the camera output with the color channels. The output of the camera doesn't have to be displayed on the screen at any point, but it must use a center of mass calculation to decide the coordinates of the paddle and pass the coordinates to the control selector(similarly to the joystick). If the paddle's center moves off the screen, it will only allow for re-entry back into the frame from the same location, so that the player won't be allowed to "teleport" the control through a different side of the screen. The output coordinates of the camera will be re-factored to match the HDMI frame size.

C. Control selector

Based on the output of a single switch on the FPGA board, the mux selects one of the joystick or camera to use as the control, saving the corresponding x and y values. It then sends the final coordinates of the paddle to the game state controller so that it can update properly.

D. Perspective Position Controller

We also plan to control the position of the camera from the player's perspective(the in-game POV). We do this with the use of 4 switches on the FPGA. Whereas the paddles can move horizontally or vertically across the entire frame display, the camera can move within a limited subset of the space. If the possible values for the paddle is (0, 1280) for x and (0, 720) for y, the possible values for the camera center position would be (400, 880) for x and (200, 520) for y. This is an additional feature that is to be implemented for the sake of showing the efficacy of ray casting. If the camera were at a single fixed location, the walls and background would remain the same the entire time, and therefore there would be no need for this feature of 3D rendering. The four switches function as "on" and "off" settings for each of the four directions, allowing for diagonal movement. If opposing(left/right or up/down) switches are both activated, the system will handle this by selecting one over the other. Across the z-axis, the camera position will remain fixed at 0. These coordinates are then sent to the 3D Ray Caster and the ball and paddle rendering where the calculations are done.

V. SPI Communication

Player 1's FPGA board serves as the main controller that initiates communication between the two FPGAs(the player

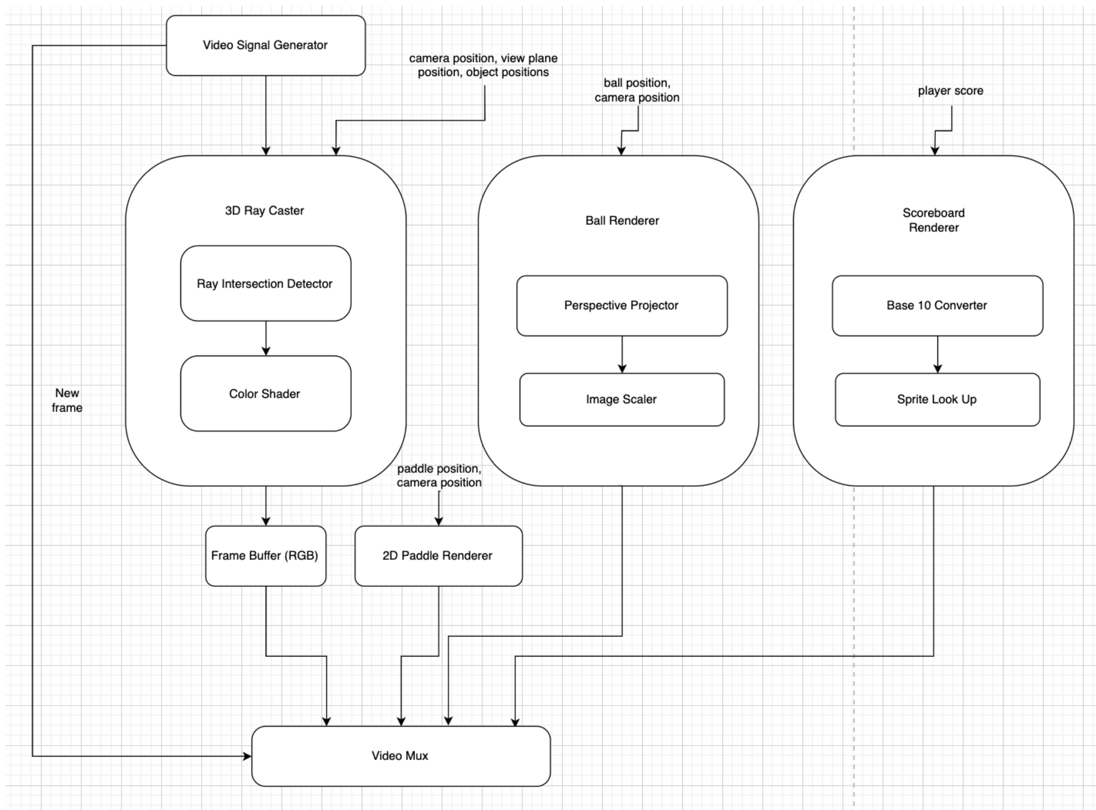

Fig. 2. Original Block Diagram for 3D and 2D Display Modules

number choice is arbitrary). The system follows the clock cycle of the data coming in and sends the data bit by bit on each clock cycle. FPGA1(main controller) pulls the selector signal low to indicate that it expects incoming data. At the same time, the data is passed from FPGA1 to FPGA2 on one line and from FPGA2 to FPGA1 on one line. The relevant information that must be shared between the board is the x and y coordinates of the paddle; these are passed consecutively with the first 10 bits corresponding to the x coordinate and the final 9 bits corresponding to the y coordinate. In the general protocol, the width of the data expected can also be specified. For each player, since the coordinates of the player's own paddle are handled through the controller-selector-game state pipeline, the coordinates of the other player's paddle are passed directly to the game state controller.

VI. 3D Display

An essential feature of 3D PONG! is its 3D display. We established a universal 3D coordinate system as shown in Fig. 5. For Player 1, the x coordinate points to the right, and the z coordinate points into the screen. For Player 2, the x coordinate points to the left and the z coordinate points out of the screen. For ease of computation, all objects have x, y, and z coordinates in the positive region.

A. Inital Attempt: Raytracing

In our initial system design, we decided to render 3D objects using raytracing, a commonly used rendering technique that simulates the behavior of light in a 3D environment to generate realistic 2D images [1]. The process of raytracing starts by casting rays from a virtual camera (a fictional point in 3D space) through each pixel of a virtual image plane (the display screen) in 3D space into the scene. The rays intersect with virtual objects. In 3D PONG!, these objects include the pong ball and the planes in the background. The 3D positions of these intersections with surfaces are calculated to determine the color and brightness of the corresponding pixels on the display screen.

Using Numpy and Matplotlib, we were able to implement raytracing with sophisticated lighting effects, shadows, and reflections as shown in Fig. 6. However, the program was computation-intensive and involved complex math operations like taking square roots and vector normalization.

When we tried to convert the algorithm in Verilog, we found raycasting difficult to implement as it involved heavy manipulation of signed fixed-point numbers. In addition, we realized that raytracing would likely result in a timing violation if we set the frame rate to be 60Hz. After evaluating the cost of implementing raytracing, we decided to switch to a simpler rendering algorithm, raycasting.

B. Second Attempt: Raycasting

Unlike raytracing, raycasting is a semi-3D algorithm that renders a scene using simple trigonometry. In raycasting, rays are cast for each column of pixels on the view plane, making

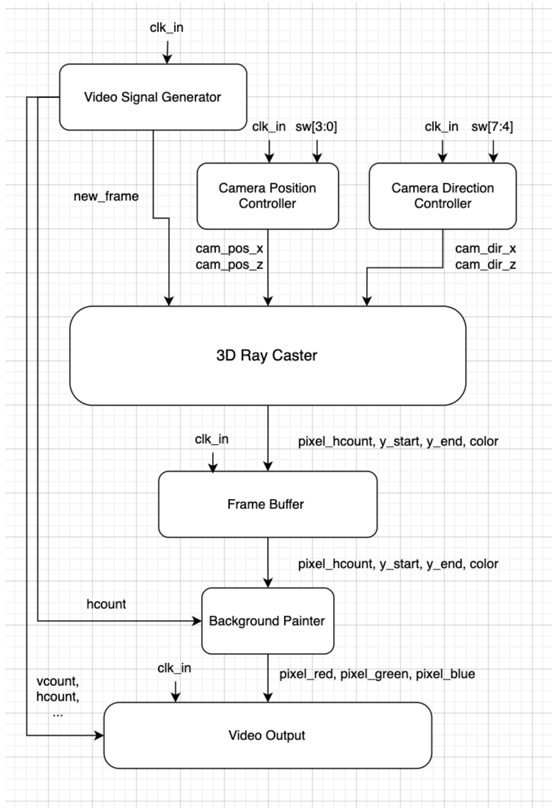

Fig. 3. Current Block Diagram for 3D Display Modules

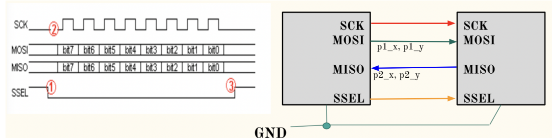

Fig. 4. SPI communication protocol

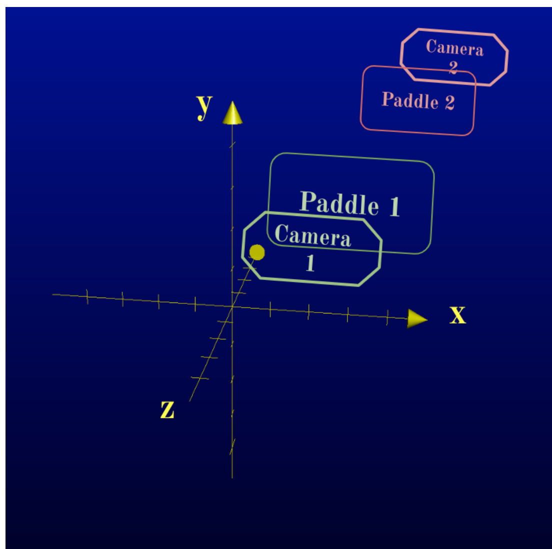

Fig. 5. 3D Coordinate System

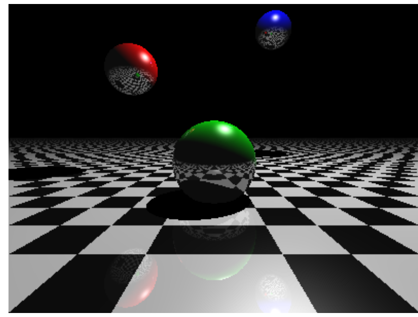

Fig. 6. Example of 3D Raytracing Implemented in Python

it faster, less resource-intensive, and easier to implement than raytracing. Although raycasting doesn't consider lighting effects like reflections and refractions, it is suitable for rendering simple game scenes in 3D PONG!.

Fig. 7 is a 3D environment of bounded walls rendered using raycasting. The cryo wall is facing the camera view plane directly, and the navy blue walls are on the sides. To implement raycasting in Verilog, we designed a 7-state finite state machine as shown in Fig. 8.

The 3D map is stored as a grid-like 2D array of 0s and 1s where 1s represent the wall and 0 is empty space. When receiving a new_frame signal, the Ray Caster starts its internal counter x range from [0, 1279] to cast 1280 rays for each pixel column on the 1280-pixel-wide screen. For each ray, the program calculates the direction of the ray and its unit distance to the edge of the grid it's in. The program then increments the ray's traveled distance until the ray hits a wall. If the ray hits a vertical wall, the color of that wall is cryo, if the ray hits a horizontal wall, the color of that wall is navy. Using the final distance the ray traveled, the program calculates the projected height of the wall. Finally, it calculates the vertical range of pixels in that column to be colored as the wall and outputs the value of x, the range of wall pixels, and the corresponding wall color to the Frame Buffer.

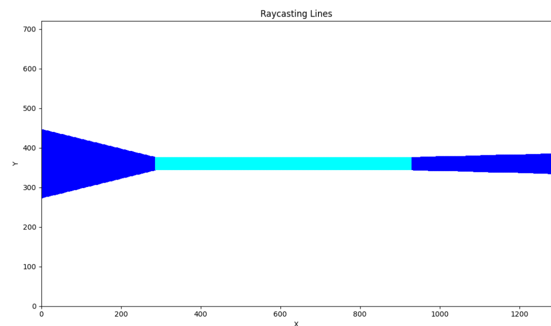

Fig. 7. Example of 3D RayCasting Implemented in Python

C. Background Painter

The Background Painter is a simple combination circuit that interprets the output from the Frame Buffer and determines the color of the pixel at ($hcount$,$vcount$). If the pixel is not

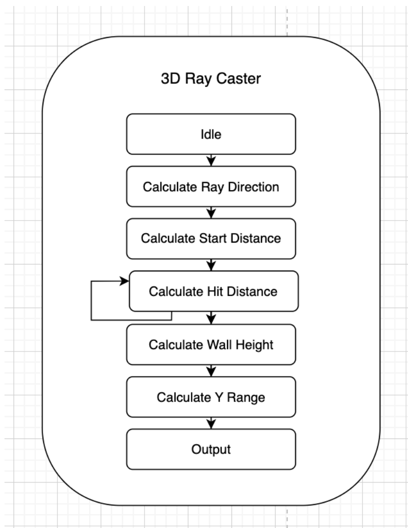

Fig. 8. FSM of the Verilog Raycaster

located in the range of wall pixels, it outputs the color black; if the pixel is on the vertical wall, it outputs navy, and if the pixel is on the horizontal wall, it outputs cryo. The color information is transmitted to the TMDS pipeline to render on the HDMI screen.

D. Camera Controllers

Finally, we integrated a module for controlling the position and direction of the "virtual" camera in the 3D environment to allow players to experience movements in the game environment like a first-person game. At each new_frame signal, camera controllers read signals from switches of the FPGA and increment/decrement the position coordinates or direction vector correspondingly. The updated camera position and direction are used by the Ray Caster to calculate the new image. There is a 1/60 second delay of the new image, but it is not observable by human eyes.

E. Display Pipeline Evaluation

A demo video of our final display pipeline is uploaded to https://youtu.be/pM02JGbsXXc [2]. Our final display pipeline renders at a frame rate of 60Hz and 100mHz clock cycle with no significant delay from camera controllers.

The pipeline uses a 32 by 1280 BRAM for storing the display information of a 1280 by 720-pixel screen. The width of the BRAM could be further optimized to be 21 bits as we can use the $hcount$ of each pixel as the address and only store the 10-bit y_start, 10-bit y_end, and 1-bit $color$.

The most noticeable area of improvement for the pipeline is its rendering quality. There are observable zig-zag patterns on the edges of the rendered side walls. This could be a result of having not enough fractional bits and rounding errors snowballed through calculations. The poor resolution of the image can be improved by using 48-bit numbers with 16 integer digits and 32 fractional digits.

Although the pipeline is not complete, its core calculation modules can be adapted to project a moving pong ball on the screen. Given a ball's 3D position, we can treat the ball as an extremely narrow wall and calculate its corresponding position and radius on the screen. The Background Painter can be modified to display the ball on top of the background environment.

F. Implementation Insights

Unfortunately, we are unable to deliver the full project as planned. We reflected on our collaboration process, and these are the key lessons we learned.

- Set up a weekly in-person work session. Working together in person is the most efficient way of communicating and solving problems.
- Have a Python/JS/C... implementation of complex modules makes debugging 100 times easier. Printing intermediate signal values will help you verify at what stage the calculation goes wrong.
- The project is only going to be harder than you think. Calculations as simple as a division can manifest into an ugly form with fractions representation, signed/unsigned numbers, response time of IP modules, etc. Plan for buffer time.
- Don't be afraid to write MORE test benches. The only way to catch a tiny bug in a massive system is to run each module through unit tests.

G. Individual Contribution

- Zitong: Sketched the overall system block diagram and ray casting pipeline block diagram. Built the 3D rendering pipeline using ray casting. Built the camera controller to allow the camera plane to perform pseudo-3D movement in the game space. Wrote Abstract, Introduction, Physical Construction, Block Diagram of Display Modules, and Project Reflection.
- Eugene: Designed game state module for handling game logic and transfer of coordinates between two FPGA boards. Built control mechanism for using either camera or joystick as player controls and designed pipeline for translating the camera, ball, paddle coordinates onto rendering. Wrote Abstract, System Block Diagram, Control Mechanism, SPI communication, and Display.

H. Appendix

- Zip file of raycasting code: https://drive.google.com/file/d/1DbcsNAVQMjozsTmUAd_hv_nDms5ZQ8Xh/view?usp=drive_link
- Python implementation of raycasting rendering on Google Colab: https://colab.research.google.com/drive/1dZQiFOILDI6GhUnRaqcBRGg2Si1r8rk9?usp=sharing
- Python implementation of raytracing on Google Colab: https://colab.research.google.com/drive/1Ssu4u3ORОdKQiQOKey7qOPLosvozyTI6?usp=sharing

REFERENCES

[1] Introduction to ray tracing: A simple method for creating 3d images. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/ray-tracing-practical-example.html. Accessed: November 21, 2023.

[2] Zitong Chen. 3d pong! partial demo, 2023. YouTube video.