Pong: 2-Player, First Person

William Navarre and Emmanuel Akinbo

Abstract

We plan to extend the lab 3 pong game to the case of two players on separate FPGAs and viewing the game on separate monitors. This involves creating a serialized communication channel to transmit messages from one FPGA to another, and poses the challenge of keeping track of which of the two identical FPGAs should serve as leader (from the perspective of gameplay, this means deciding which way the ball should go first). We also plan to have a 1-point perspective “3D” view of the game which carries its own difficulties due to the complexity of implementing the trigonometry involved.

Introduction

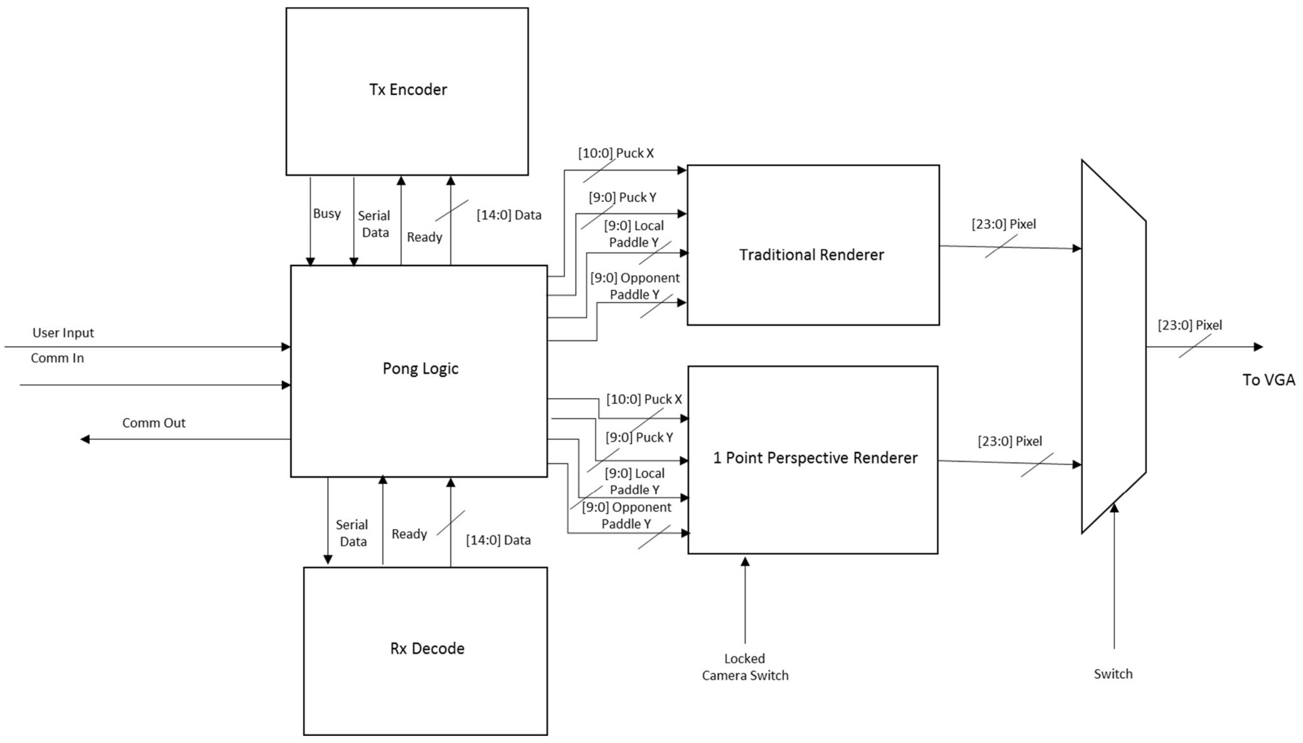
We break down the pong game into a main module, which keeps track of all the state of the game, two communication modules (input and output), and two graphical rendering systems that allow for 1-point perspective and standard view.

The details of serialized communication are dealt with by the transmission and receiver modules, making it easy for us to easily change the implementation without changing each module's extremely simple interface. This is what will allow us to easily prototype different physical layer communication channels.

The bulk of the work is really handled in the main pong module so that the visualization modules can be stateless (module pipelining which may be necessary for the 1-point perspective view).

The 1-point perspective view creates several challenges of its own: mostly, how to implement the trigonometry, and how to implement (or avoid implementing) division.

Block Diagram



Modules

Pong Logic

The Pong logic is responsible for keeping track of the position of every object on the board. At a specific time on each screen refresh, it recalculates the position of each object as appropriate: when necessary, the ball's position will be updated to simulate motion, and the user's paddle's position will be updated when required by a button press. At a separate time during the screen refresh process, the module considers whether any “bounces” are appropriate and acts accordingly.

It also deals with managing what is communicated over the wires and making use of that data. At a separate time in each screen refresh, it sends over its belief about ball velocity (horizontal and vertical velocity, signed 5 bit numbers), the ball's horizontal (11 bit unsigned) and vertical (10 bit unsigned) positions, and the local paddle's current position (11 bit unsigned, for simplicity). Game state information is also transmitted to communicate losses and successful paddle bounces.

In all cases, when an FPGA receives data about the remote paddle, it updates that position information in state so that the information can be displayed to the monitor. When a ball is actually coming toward the local paddle (so the local player needs to be ready to hit it!), the FPGA actually ignores the information. Only when the ball is going away does it use that information to update the position of the ball so it can be drawn to the screen, and in this case does not otherwise recalculate a ball's position itself, until it receives a message that a bounce has happened, along with the usual data: at that point, it uses the velocity and position information it received to smoothly take over calculations.

Transmission and Reception

The job of the transmission and reception modules is to serialize and de-serialize data. A fixed width bus on each allows for un-serialized (parallel) information, while a line (connecting the FPGAs) provides a place to read and write serialized information.

The transmission module accepts a bus input which is saved into state when “ready” input is asserted. Then, onto the serialized line, a preamble followed by the zeros and ones are written in series.

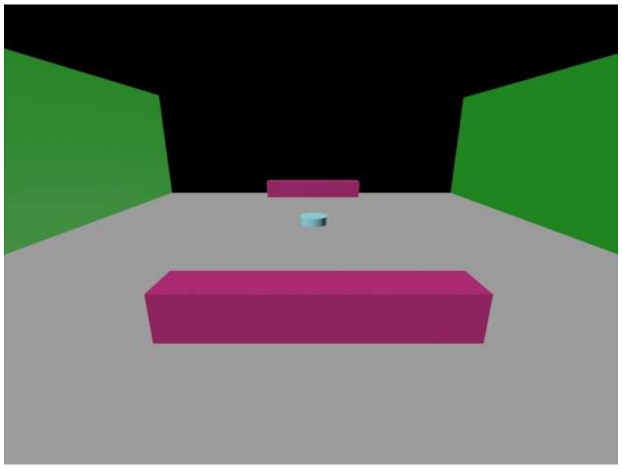
On the receiving end, preamble detection is implemented and then the data received is written to internal registers. When the entire fixed-size message is read into the internal registers, their data is transferred to registers exposed through the output pins.

Traditional Renderer

Our traditional renderer will function similarly to the typical implementation of pong. From the output of the pong logic module, the traditional renderer will receive the location of the puck, the local user’s paddle, and the opponent’s paddle location. Using this data, it will select the pixels to be lit accordingly and send out the data to the MUX selector for VGA output. The output from this module, when selected on the MUX, will output the traditional top down gameplay to the VGA monitor.

1 Point Perspective Renderer

This module will produce our new take on the traditional pong game. Instead of creating the top-down view normally seen in pong, this module will create a perspective output view of the game from a location slightly above and behind the local user’s paddle, such as in the image that follows.



This module will take in the same resources as the traditional renderer, namely the location of the local user’s paddle, the location of the puck, and the location of the opponent’s paddle. However, this module will also take an additional input of a switch on the FPGA to toggle having a stationary camera or a camera that follows the location of the local user’s paddle.

To create this output, the module will first assign height values to the different objects in the scene (pucks, paddles, walls) via default parameters to create a 3D mapping for each of the objects. Then, using 3D to 2D projection algebra and the location of the camera, it will calculate the relative sizing for each object to be displayed on our 2D monitor screen. The algebra in use will be as follows:

$$\begin{bmatrix}\mathbf{d}_x\\ \mathbf{d}_y\\ \mathbf{d}_z\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\theta_x) & \sin(\theta_x)\\ 0 & -\sin(\theta_x) & \cos(\theta_x)\end{bmatrix}\begin{bmatrix}\cos(\theta_y) & 0 & -\sin(\theta_y)\\ 0 & 1 & 0\\ \sin(\theta_y) & 0 & \cos(\theta_y)\end{bmatrix}\begin{bmatrix}\cos(\theta_z) & \sin(\theta_z) & 0\\ -\sin(\theta_z) & \cos(\theta_z) & 0\\ 0 & 0 & 1\end{bmatrix}\left(\begin{bmatrix}\mathbf{a}_x\\ \mathbf{a}_y\\ \mathbf{a}_z\end{bmatrix}-\begin{bmatrix}\mathbf{c}_x\\ \mathbf{c}_y\\ \mathbf{c}_z\end{bmatrix}\right)$$

$$\begin{aligned}\mathbf{b}_x &= \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_x-\mathbf{e}_x\\ \mathbf{b}_y &= \frac{\mathbf{e}_z}{\mathbf{d}_z}\mathbf{d}_y-\mathbf{e}_y\end{aligned}.$$

Where:

- $a_{x,y,z}$ is the location of the object being projected
- $c_{x,y,z}$ is the location of the camera viewing the object
- $\theta_{x,y,z}$ is the orientation of the camera viewing the object
- $e_{x,y,z}$ is the location of the viewer relative to the display surface
- $b_{x,y}$ is the projection of the image on the monitor

Resources

Our materials list is as follows:

- Two Virtex-2 FPGAs (6.111 lab FPGA)
- Two IR LED transmitters
- Two IR receiver chip
- 1k ohm resistors
- 47 ohm resistors
- 3.3 uf capacitors
- 2N2222 BJT transistors
- 0.1 uf capacitor
- Ultrasonic transmitter and receiver

Timeline

By the end of each week, we plan to complete the following elements of our project:

- 11/7 - Communication Prototype
  - Have a basic wired prototype of our transmission and reception working
- 11/14 - 3D module
  - Have the one point perspective view working
- 11/21 - IR Communication
  - Implement our wireless transmission and reception including creating a handshake and the real-time data transmission

- 11/28 - Sound Communication module, Second puck
    - Create the implementation of our sound communication module
    - Add in the option to play with a second puck
        - Note, both of these are stretch goals
- 12/5 - Debugging
- 12/12 - Debugging