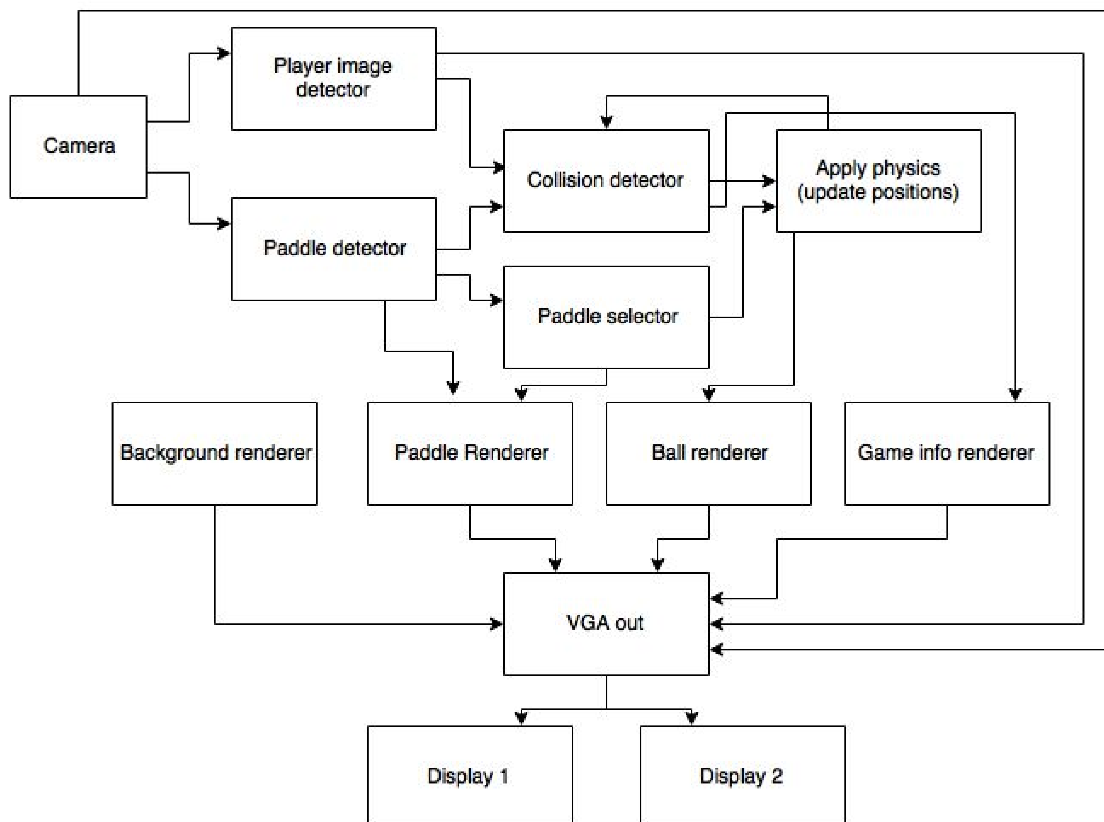
Live-Action Pong

1. Summary

We plan to program a pong or tennis-style game that incorporates both the physical and virtual world. Each player will face a monitor, with a green screen on one of their sides and a video camera on the other. They will hold a paddle, which will be marked with bright LEDs so that it will be easily identifiable in image processing. The monitor will display the virtual playing field, displaying a side view of both of the players. The background and paddles will be replaced with sprites. The players will each hit a ball back and forth using their paddle, attempting to get the ball past their opponent. Additionally, they will be able to use buttons on their paddle to launch attacks, which the opponent must try to dodge entirely. The physics of hitting a ball will be simulated using either differences between frames or IMUs to recognize the velocity and angle of the paddle. Sounds and other effects may be added as opportunity permits. A stretch goal would be to allow the user to switch between different types of paddles that have different effects, for instance one shaped like a lacrosse stick that can catch and hold the ball, or one shaped like a golf club for additional velocity.

2. Block Diagram



3. User Inputs and Necessary Hardware

a. NEXYS 4 Development Board

b. Video camera

The video camera will record digitally at 640 x 480 (480p video). The output will be transmitted to the FPGA development board over USB, which will be the physical input into the FPGA.

c. Monitors (2x), VGA splitter

Two desktop monitors will be needed to display the state of the game to each player. Because of the side-angle view, the same screen can be shown to both players without alterations. We intend to use a VGA splitter to display the same image.

d. Green screen

Two 5’ by 7’ green screens will be mounted on a wall, oriented so that the short edge of the individual screens will be on the floor. Combined, the green screens will provide us with 10’ by 7’ of space for the players to move about in. The screens may be separated slightly so that we aren’t wasting green screen behind the monitors.

e. Physical paddle with LEDs, potential IMUs

A physical controller will need to be built for the players to control their on-screen paddle. A stick about a foot long, not including the handle, will be wrapped with LEDs in two locations, one at the base and one at the end. The LEDs will be wrapped all the way around the device so that the bright lights will be visible to the camera no matter the angle. A series of buttons may also change the color of the lights, allowing for an attack to be signaled if this feature is implemented.

4. Software Modules

a. Human from Green Screen (Mike)

For every frame provided by the camera this module will distinguish which pixels represent the player and which pixels represent the background (background pixels will be almost entirely green). The arithmetic operations here are simply greater-than comparisons between the pixels HSV values and an HSV value threshold greater than which would indicate that the pixel is green. The module does this check for every single pixel input from the camera and stores the result (0 for not green, 1 for green) in a buffer so it is easy for later modules whether a pixel represents a player or is the background. Using this output the collision

module has the information it needs to determine where the player is for collision checks, and the VGA output module can determine when to use the camera pixels as output in order to show the player him/herself on the game field.

b. Paddle Position / Angle / Velocity Detector (Nick)
This module will take input from the video camera and extract the location of the paddle. The player-held paddle will have several LED's along its edge, which will be visible to the camera regardless of any rotation of the controller. Provided that the LEDs are of sufficient brightness, the magnitude of light coming from them will be an outlier when compared to background light. The module will then find a central point in each bright blob. Once these two points are known, finding the current angle of the paddle will be easy. The current position and angle will be outputted from this module. As a stretch goal, we will attempt to find the current velocity as well as position, so that we can change how much force is applied to the ball. We can take the two LED positions from the current and previous frames to find the difference between them, allowing us to approximate the velocity.

c. Collision Detector (Mike)
    i. Ball-paddle: Uses the position and angle data from the paddle detector to determine the extents of the paddle, then uses the position of the ball to see if the ball has hit the paddle. Updates the ball velocity appropriately if so. If we have time to implement our stretch goals, it will also use the angle of the paddle to determine in which direction the ball will bounce off, and also use the velocity of the paddle to affect the speed of the ball accordingly.
    ii. Ball-wall: A simple check to see if the ball should bounce off the top or bottom wall, and if the ball has hit a wall behind a player in which case the player should lose a life.
    iii. Attack-player: If we achieve our stretch goal we will need to also check for collision between a launched attack and the player. This is not too difficult to achieve given the output of the player detector already tells us where the player is in the game space. Complexity arises when we need to handle a variable number of attacks on screen (since they are player generated and disappear once they hit a player or go offscreen).

d. Physics Simulation (Nick)
   This module will take the current position, angle, and potentially velocity of the two paddles as inputs. The module will also receive notices from the collision detector. When there is no collision, the module will simply update the positions of the ball and other moving objects (i.e. spells). When there is a collision between the ball and the paddle, the ball's velocity will be updated. This module will have a spectrum of features from basic functionality (negate horizontal velocity) to more complex ideas. The angle and velocity may be used to provide a better physics simulation, so that harder hits move the ball faster and angling the paddle will cause the ball to travel in the direction you expect it to go based off classical mechanics. Another feature which we might add is to allow for a catch and release, similar to a lacrosse stick. Regardless of the complexity of our implementation, the output of the module will be the most up-to-date position of the ball.

e. Image Rendering (Mike)
   i. Ball: Using the pipelining techniques in the final LPset it is simple to render a circular ball. Game difficulty could be increased by making the ball mutate or behave differently throughout the game.
   ii. Paddle: Using information from the paddle detector (the paddle's location and angle), and the paddle selector (what type of paddle is currently selected), this module will decide where and at what orientation to render the appropriate paddle sprite. Sprites could be generated with simple equations for rectangles and circles, or could be pre-loaded into the FPGA's memory.
   iii. Background: Lots of leeway here, we can make the background as complicated or simple as we want. Ideas include having the background react to a player hitting the ball with their paddle, or changing colour to make the ball harder to see and thus the game more challenging. One feature the background must have is some sort of platform graphic so that the players know the area in which they must stay (this will reflect the horizontal extents of the green screen)
   iv. Game logic: Uses the output of the collision detector module to determine if a player has missed the ball and thus should lose a life. Displays lives remaining for both players. Could also use the information from the player detection module to see if a player has walked out of bounds.

5. Timing

The VGA format will expect a refresh rate of 60Hz, meaning that each cycle must be completed in 16.67 ms. Using our 2.7 MHz clock, this allows for 45,000 clock cycles per refresh cycle. This will not provide enough time for individual pixels to receive their own 2.7MHz clock cycle, as there are 307,200 pixels on our screen. Image processing and rendering calculations must therefore be done in parallel, and output to VGA will need to be operated on a faster clock cycle.