6.111 Final Project – Intuitive Video-Game Controller
Fall 2006
Eddie Fagin, Irene Fan

Our project consists of two portions: the controller and the game. These parts will be developed separately and are mostly abstracted from each other. The controller can be tested using the video output and hexadecimal display, and the game can be tested using the FPGA button inputs. The controller will take camera input, filter the data, and generate simple directional information each frame as a three-bit output to the game. The game will take this information and use it to drive an entertaining game from the Atari-era. Our goal is to make a classic game better through a more interactive controlling mechanism.

Eddie will primarily develop the controller interface. This portion of the project processes the camera input and converts that data into directional pulses for the game. First, we will borrow and modify modules from previous terms; these include the ADV7185 interface and a ZBT read and write mechanism. These modules, originally used to display real-time black-and-white video footage, will be adjusted in numerous places to generate a real-time color (RGB) image. Initially, the camera data is stored as 30-bits/pixel, representing the three ten-bit values of Y (luminance), Cr, and Cb (both chrominance). The black-and-white project stores and displays only the luminance data for each pixel, and throws out the chrominance values. These modules need to be adjusted to store all three values. First, the NTSC decoder will be adjusted to convert the camera data to 24-bits/pixel, because the project should not need more than eight bits per value. Then, the storage mechanism will be altered to retain one 24-bit value in each line of ZBT memory, instead of four 8-bit luminance values. Due to this conversion, only a quarter of the values will be stored in memory; the current module is programmed to advance lines after four pixels. However, since this image is not displayed on the screen during normal program operation, a fine-tuned image is not necessary and potentially inefficient.

These steps produce a stream of YCrCb data from the camera. The project will eventually analyze all high-intensity green lights (green LEDs and any noise from the overhead fluorescent lighting), so the second phase of the controller interface will convert the YCrCb data into a more manageable form. We borrowed the YCrCb2RGB converter from the Xilinx toolkit, and will use this module to create a 24-bit RGB array for each 24-bit YCrCb from the ZBT memory. This RGB data is passed through a filter that removes all red, green, and blue values below a certain high cutoff point, close to 255. Of the data that passes this cutoff, only the strongest red, green, or blue value passes through the filter. This process removes most of the image data, allowing only the brightest reds, greens, and blues to reach the next part of the project. Since we will only be handling a bright green LED, we will ignore the red and blue data. However, if we wanted to add other colored LEDs, the process would not require much additional effort.

The green pixel data passes to “Phase 3” of the controller interface: the analysis. The first module will calculate the effective center of pixel “mass,” and attempts to remove any anomalies caused by noise or other external factors like the overhead lighting. The project requires a robust and consistent game input mechanism, so all of the

analysis modules will require a great deal of time and effort to implement correctly. Once the center of mass in each frame has been calculated, another module (COMvelocity) will calculate the velocity (speed and direction) of this point, based on what it knows about the previous point's position and velocity. In order to simplify the code, and to meet the specifications of the game, "speed" can take on one of three values (slow, medium, and fast), and "direction" can be up, down, left, or right. Therefore, this module has 12 possible outputs to the final module, gameDirection, which converts the velocity information to a series of pulses that the game uses as input. These pulses are only directional; the speed only affects the frequency of these pulses.

For debugging purposes, the processed camera data will be displayed on-screen, and unless this function severely affects the game functionality, the camera image should be readily accessible in the final version of the project. The entire controller section of the project will only use a few buttons on the FPGA, mostly for debugging or adjustment purposes. The switches, LEDs, and hexadecimal display will remain available for the game. For equipment, we will need a color video-camera (already acquired) and a wireless, button-activated LED, probably attached to some sort of controller. This controller will not be complicated; in all probability, it will just be a small block of wood with a battery attached.

Irene will focus her efforts on the game portion of this project. The game will be our version of Atari's "Super Breakout" (1978). In Super Breakout, there are rows of bricks lining the top of the screen and a ball that bounces off the side and top walls, breaking bricks whenever it ricochets against them. A paddle is located at the bottom of the screen, and it can be moved left and right to deflect the ball; if the paddle misses the ball and the ball touches the bottom of the screen, the game is over. The user's main motive is to break all the bricks on the screen, and the bricks will be a variety of colors to indicate how many times they must be hit before they break and disappear from the screen.

The initial portion of this project will be similar to the Pong game made for Lab 5. The blob module will be reused to create the paddle, and for the ball a similar module will be made to draw a circle of a specified color and radius. The colored pixels in the ball are generated from calculations done on the hcount and vcount inputs instead of being loaded in from a .coe file so that the radius of the circle can be changed with relative ease. So, for this ball module, the inputs and outputs are the same as those for blob: x and y, which are the coordinates of the top left of the corner of the box enclosing the circle; hcount and vcount, the coordinates for the pixel the XVGA is looking at; and pixel, the color the current pixel should be as determined by the module. Calculations in this module are limited to checking whether or not hcount and vcount are in the circle, via the equation for a circle.

The bricks, although rectangular, will not each be an instance of blob. Instead, for the 2 or 3 levels of the game, the initial configuration of bricks will be predetermined by a .coe file that will be loaded into the FPGA. We will consider the screen for our game as a grid, with each block in the grid being the size of a brick. Each block has information about the color of the brick in that location, the number of times the brick needs to be hit before it disappears from the screen, and whether or not the brick should be displayed on the screen. The bottom area of the grid where there were no bricks to begin with are marked in the memory as blocks that, from the start, should not be displayed.

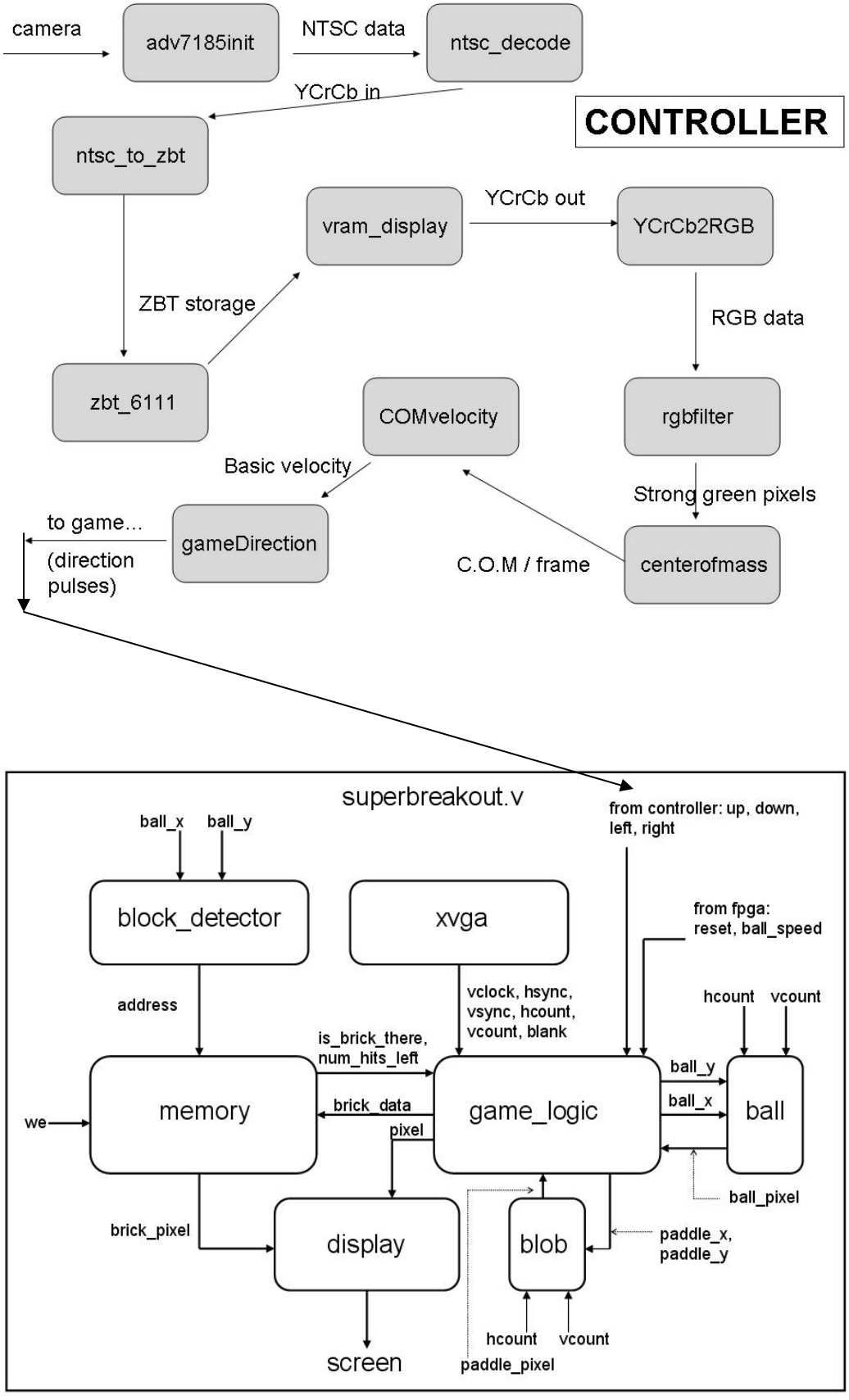
At each stage of the game, the game module calculates where the ball is projected to travel, and then these coordinates are fed into the block detector module. The block detector module calculates which block in the grid (and therefore which memory address) the particular pixel lands, and then passes this address to the memory. The memory checks to see if the selected block in the grid has a brick—if so, it decrements the number of times the brick still has to be hit before it is no longer displayed, and also notifies the game module so that the game logic can have the ball "bounce off" the brick. If there is no brick there, then the game logic knows the ball does not change direction.

At the same time, the display module takes the pixel outputs from the blob and ball and also cycles though the memory module to draw the ball, paddle, and bricks on the screen.

Besides the blob and ball modules, our version of Super Breakout makes use of the xvga module from Lab 5, and also has modules for the game logic, memory, block detector, and display. The xvga module is for displaying 1024x768 pixels, so it runs off of the 65MHz clock calibrated from the FPGA's 27MHz clock crystal.

The game logic takes the XVGA signals, the up/down/left/right inputs from the controller half of the project, the reset and ball speed signals from the FPGA, and the brick information (whether there is a brick in this particular location) from the memory; with all this information, it decides where the ball should be going at each step of the game. The game module thus outputs updated brick data (decrementing the remaining number of hits needed to demolish a certain brick, or changing whether or not the brick should be displayed) and the write enable signal for the memory; it also outputs the new x and y coordinates and the pixel signal for the paddle and the ball.

The block detector module takes the ball_x and ball_y signals from the ball module and generates the memory address for the block in the grid that encloses the specified pixel. These calculations will be quick arithmetic operations or even look-ups based on how many blocks the screen is divided into. The memory module is a dual port memory and accesses the given address location as well as outputs data about that block in the grid. From the memory, the display module receives information about the colors for each block in the grid (black if there is no brick or else the color of the brick if it exists); the game logic module receives signals indicating if there is a brick in the location of ball_x, ball_y and if so, how many more hits the brick needs before it should disappear. The memory module also has a write enable and data-in signals controlled by the game logic for updating the status of the game. The display module loops through the addresses of the memory to get information about generating each block of the grid and combines this information with the pixel outputs from blob and ball to update the screen and display the progress of the game.


camera
adv7185init
NTSC data
ntsc_decode
YCrCb in
CONTROLLER
ntsc_to_zbt
vram_display
YCrCb out
YCrCb2RGB
ZBT storage
RGB data
zbt_6111
COMvelocity
rgbfilter
Basic velocity
Strong green pixels
to game...
(direction pulses)
gameDirection
C.O.M / frame
centerofmass
superbreakout.v
ball_x
ball_y
from controller: up, down, left, right
block_detector
xvga
from fpga: reset, ball_speed
address
vclock, hsync, vsync, hcount, vcount, blank
hcount
vcount
is_brick_there, num_hits_left
ball_y
ball_x
we
memory
brick_data
game_logic
ball
pixel
ball_pixel
brick_pixel
display
blob
paddle_x, paddle_y
hcount
vcount
screen
paddle_pixel