**CS152B**

**Final Project: Motion Detection and Tracking**

Section 1A

TA: Babak Moatamed

Group 9: Jeannie Chiem, Binyi Wu, Nick Rivera

# Overview

In this project, we used the FPGA to implement a motion detection and tracking system. We detect the motion of a bright object using a camera, and we track the movement of that bright object. Based on the motion of the object, we draw the object’s path and move a little on-screen robot in the direction of that motion.

# Hardware Component

We used the FPGA board and a camera for this project. Initially, we planned to make use of the camera to control an iRobot through an RS232 serial cable, however, due to unforeseen complications and lack of time, we decided to focus solely on the camera.

Camera

Using the Vmod camera configuration provided to us from the class on Xilinx Platform Studio, we generated a bit file for the MicroBlaze soft processor that allowed us to read visual input from the camera and write it to a monitor through HDMI.

Complications

For the hardware component, we ran into many issues when including the camera peripherals into the project. The program reads the visual camera data written to the FPGA block ram in order to draw the image on a screen. This, however, led to issues with the FPGA memory where the program would try to read and write data from memory that was outside the camera input range due to a lack of space. This would cause the camera to fail setting up. We managed to get the camera to run by changing parts of the memory region mapping on the project’s linker script to allow the camera more space to write its data.

Though we had the camera working, pairing different peripherals with the camera was another issue that we faced. The project originally included an iRobot component that would be controlled through a UART peripheral serial connection based on the visual input from the camera. We managed to get the iRobot working on its own using the same MicroBlaze processor; however, when added to the camera project, the iRobot serial cable would no longer transmit data at all.

After spending much time trying to get it working, we decided to simplify our project to represent the robot visually and include switches to change the different views on the screen. However, similar to the iRobot, the switches would work on its own project, but when added with the camera, would fail to work. Looking through the memory itself using the debugger showed that the switch pins failed to be read at all when used with the camera.

Due to these issues and lack of time to fix them, we decided to rely just on the camera itself for the project.

# Software

We implemented our program’s logic in software. This made it easier to debug because we can step through C code using the debugger.

Camera Initialization and Configuration

First, we need to initialize the camera. This has two main steps. The first step involves many calls to XIo\_Out32 with different arguments. Each call of this function places a different value in memory at a different address. We call this function repeatedly because the camera needs many parameters to configure itself, and we need to place each parameter in a memory location that the camera can find.

Once we place these parameters in memory, we need to call CamIicCfg and CamCtrlInit. There are two cameras mounted on the PMOD camera, and we only need one. This means that we only make one call to CamIicCfg and one call to CamCtrlInit. We pass into these functions the size of the image that we want, in this case 640 by 480 pixels. After CamCtrlInit finishes, our camera is configured. We then clear the screen by drawing white to every available pixel.

In between each call to XIo\_Out32, CamIicCfg, and CamCtrlInit, we draw a different color on the screen. This lets us know when each function has finished running. We can draw colors and shapes to the screen by drawing them pixel by pixel. To draw a color, we find the x-coordinates and y-coordinates of where we want to draw. Then, we find the RGB value of the color that we want to draw. Finally, we use XIo\_Out32, going coordinate by coordinate, and placing the color value of that coordinate in memory.

Grid-based Brightness tracking

Next we detect the motion of a bright object. In an infinite loop, we read the incoming image using XIo\_In16. We detect motion by splitting up the incoming camera image into a 10-by-10 grid. This means that we start on the upper left of the image, and scan a 64-by-48 block of pixels, pixel by pixel.

Each pixel is 16 bits large. The upper 4 bits are meaningless for us. The lower 12 bits are what hold the color information for the pixel. We can split the lower 12 bits into 4 bits for red (the upper 4 bits), 4 bits for green (the middle 4 bits), and 4 bits for blue (the lower 4 bits). Most people use 32 bits for color: 8 for red, 8 for green, and 8 for blue. 32 bits allows more colors to be represented. In this case, because we are using only 12 bits, we can only detect a limited range of colors.

Once we read in the data for a pixel, we look at its red component. We can get the red component from the 16 bit pixel by first shifting it over by 8 and then bitmasking it to see only the lower four bits. Once we have only the red component, we check if it is equal to 15, i.e. all the bits are 1s. This lets us detect whether a pixel is bright red to white, which is our criterion for checking whether an object is white. If the pixel passes this check, we add it to a running counter for our current 64-by-48 block. Once we finish scanning the current 64-by-48 block, we move on to the next block. We store the running counter using a 2D array, indexed by the row and column of the block.

Once we finish scanning all the blocks, we compare the running counter for each block by looping through our 2D array. We loop through the array to find the block with the highest count, which we then mark as the brightest block. We record the row and column of the brightest block for the next step.

The next step of our tracking program is to display the path of the movement. Here, we take advantage of the fact that whatever we write into memory stays in memory. So, if we want to display a history of the object’s movement, we do not have to keep track of all the previous brightest color blocks. What we do is each time after scanning, we draw the current brightest block in red on the top left of the display, and the previous brightest block in green. All previous green blocks stay in memory and displayed on the screen, so adding a new green block leaves the old ones in place. This is how we see a trail of where the object has been: all previous green blocks remain in memory. The camera reads from this memory and displays it without our intervention.

Our next step is to draw a grid in the bottom right of the screen that clearly display in which direction the robot will move. We again divide this section of the screen into a 10-by-10 grid. In each grid square, we will draw a small 5-by-5 pixel square. The square will be red if it is our current brightest block, green if it is the previous brightest block, and black otherwise. Unlike in the top-left of the screen where we show the path of the motion, here we only show the current brightest block and the previous brightest block. This difference means that we have to manually clear out memory when a 5-by-5 square is no longer the previous or current brightest block. Therefore, every time the brightest block changes, we loop through all 100 squares and update the color in memory to either green, red, or black.

Robot Movement and Other Features

Finally, we draw our on-screen robot. Our on screen robot is a 9-by-9 square that moves around in the bottom-left of the screen. We find the velocity of the robot by finding the difference between the brightest block and the previous brightest block. We subtract the x-distance between these two blocks to find the robot’s x-velocity, and we subtract the y-distance to find the robot’s y-velocity. We also check to make sure that the robot does not drive off the screen by checking that its updated position is not out of bounds.

Drawing the robot is a little different than drawing our brightest and previous brightest squares. Because the robot, which is magenta, frequently changes position on a white background, we need to white-out the previous location of the robot. If we didn’t, there would be a smear of magenta as the robot moved because the robot’s color remains in memory until we clear it. So, to draw the robot we loop through each pixel in a 640-by-240 portion of the bottom-left of the screen. Then, for each pixel we check whether the pixel is in the robot’s current location. If it is, we paint that pixel magenta. If it isn’t, we paint it white. An alternative approach would have been to paint all the pixels white in this portion of the screen, and then paint the magenta robot on top of it. However, this approach causes flickering on the screen as the robot moves.

After we draw the robot, our program loops back up and begins the process of finding the brightest square again.

The last feature that we added was the ability to see what colors the camera is filtering out. After the bright object moves over 25 out of the 100 squares in our 10-by-10 grid, we switch modes to our “robot vision” mode. In this mode, in the top-left of the screen we show the camera’s feed. But instead of the normal camera feed, we show only the pixels that have passed our brightness criterion by having all 1s as its RGB value. By looking at this view of the camera feed, we can see which pixels our program is filtering out. This helps us see the shape of the bright object because only the bright object is kept in the image and all dark pixels are filtered out.

# Challenges and Solutions

Color Encoding

As mentioned above, each pixel is 16 bits large. Based on the width, we first assumed the pixel to be in u16 color encoding, which has the upper 5 bits for red, middle 5 bits for green and lower 6 bits for blue. However, after experiments, the actual mapping of the color in the pixel is the 11 - 8 bits for red, 7- 4 bits for green and 3 - 0 bits for blue.

Initially, we aimed to do motion tracking based on color. We set red objects as our target, but the camera quality resulted in washed out colors around the center of its lens, which gave an inaccurate color result. Moreover, limited by the 4-bit range of each red, green and blue component, the filtered result is not desirable. We tried to convert the color to the HSV model because hue, saturation and value represent the color in a way similar to human perception. However, the 4-bit width color component is a performance bottleneck. The common conversion formula to the HSV model that we found was for the RGB model, of which each component has a value in the range of [0, 255]. Our model only has a range of [0, 15], so we could see many human perceivable differences between two pixels that map to the same HSV value. Thus, we decided to use brightness as our detection mechanism, as it provides more reliable performance.

Complexity vs. Latency

Our task requires real-time processing of the camera input, which means that we need to keep scanning through every pixel within an infinite loop. Real-time requires a low latency output of each iteration.

First, a resource that limits our program complexity is memory. Like a CPU, whose cache has the lowest latency and smallest size, and the external disk has the highest latency and largest size, our FPGA also has its memory hierarchy. As the camera configuration had already taken up a large memory space, we could not store too much data or extra cycles will be needed for data access. To maintain low latency, we use a 10-by-10 grid for a 640-by-480 actual image so that no extra arrays will be created.

Second, the size of the problem also limits the program complexity. For each iteration, we’re guaranteed to scan through each pixel, which equals 640\*480 iterations. With a similar setting, suppose we want to increase the grid size from the current 10\*10 to 20\*20. This equals at least an extra 300 iterations, which is not trivial. Because the FPGA is not designed for providing multithreading or prefetching to hide the latency for these computations, we could see that the output of such an algorithm has a high latency when compared to simply reproducing the camera input.

Third, the FPGA architecture is not good at dealing with divergent programs. Because in a hardware implementation there are no instructions on board but actual circuit units are programmed to do the computation, an if-statement might cause a huge overhead in terms of routing and memory arrangement. With fewer conditional statements, however, the complexity is limited.

We finally chose to prioritize latency because we believed it’s more reasonable for dealing with camera inputs to control robots.

JTAG Error

Our team’s FPGA had a really fragile USB port, and it frequently reported JTAG errors. One possible solution is to go to the JTAG configuration and change the setting from automatic detection to the specific JTAG connection being used. This did not help us because it was the physical connection of the port itself that was a problem.

# Reference

<https://stackoverflow.com/questions/3018313/algorithm-to-convert-rgb-to-hsv-and-hsv-to-rgb-in-range-0-255-for-both>

<https://www.mathworks.com/matlabcentral/answers/230051-filter-image-such-that-only-certain-colours-show>