Lightsaber Training

Landon Carter (lcarter), Rachel Yang (rsyang), Linda Zhang (lolzhang)

6.111 Final Project Report | Fall 2016

Abstract

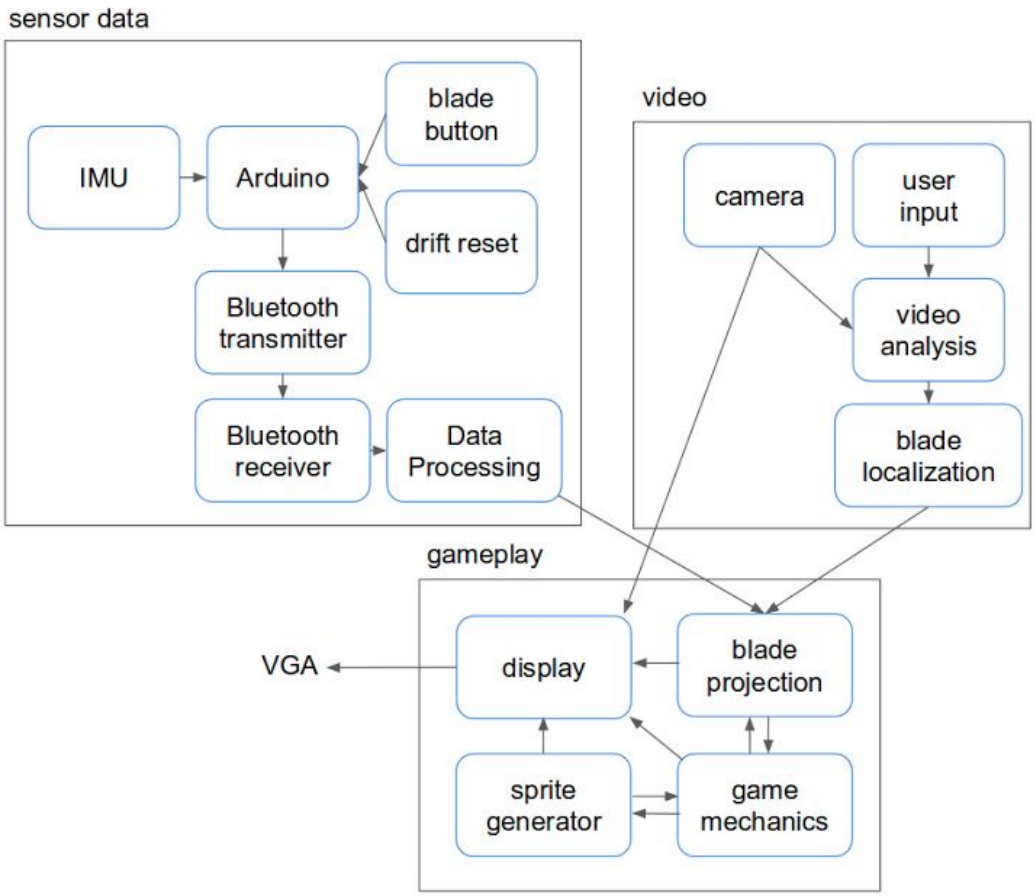
Lightsaber Training is an augmented reality game, where the user holds a device representing a lightsaber handle, while a lightsaber blade is projected on-screen. The user can then swing the handle around and attempt to hit as many objects on-screen as possible with the blade in a given time frame. This is accomplished via image tracking from a video feed, a wireless IMU embedded in the lightsaber handle, a button to extend and retract the blade, and generated sprites. The project's three primary modules are the sensor data, which includes wireless accelerometer and gyroscope data and button inputs; video, which includes camera input and image thresholding; and gameplay, which includes blade projection, sprites, and game mechanics.

Table of Contents

Overview

We implemented an augmented reality game in which a user, wielding a lightsaber handle, attempts to hit objects displayed on-screen. Our project uses image tracking from a camera connected to the FPGA labkit and IMU data transmitted via bluetooth from the handle to project the lightsaber blade on-screen, overlaid against the real-time video stream from the camera. A button on the handle also lets the user extend and retract the blade. The user's movements are continually tracked so that the projected lightsaber blade moves with the handle. Gameplay includes sprites that appear on-screen and a scoring system that tracks when the user hits a sprite with the lightsaber blade within a given time frame.

High-Level Block Diagram

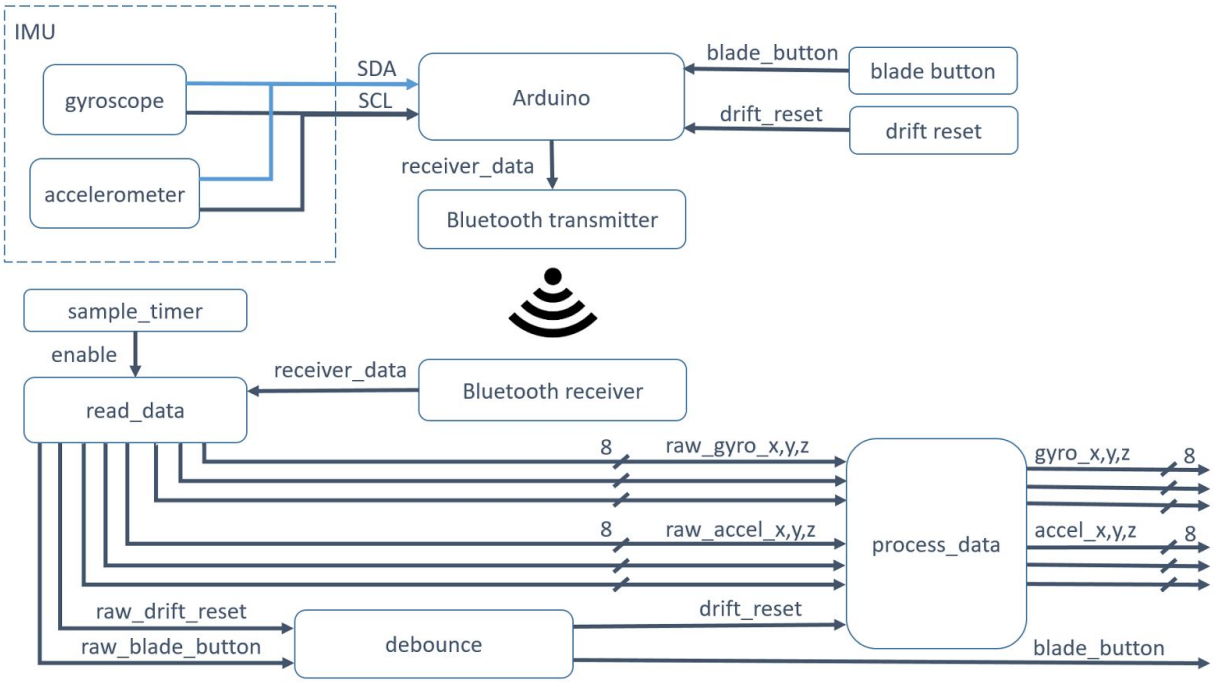


Part 1: Sensor Data (rsyang)

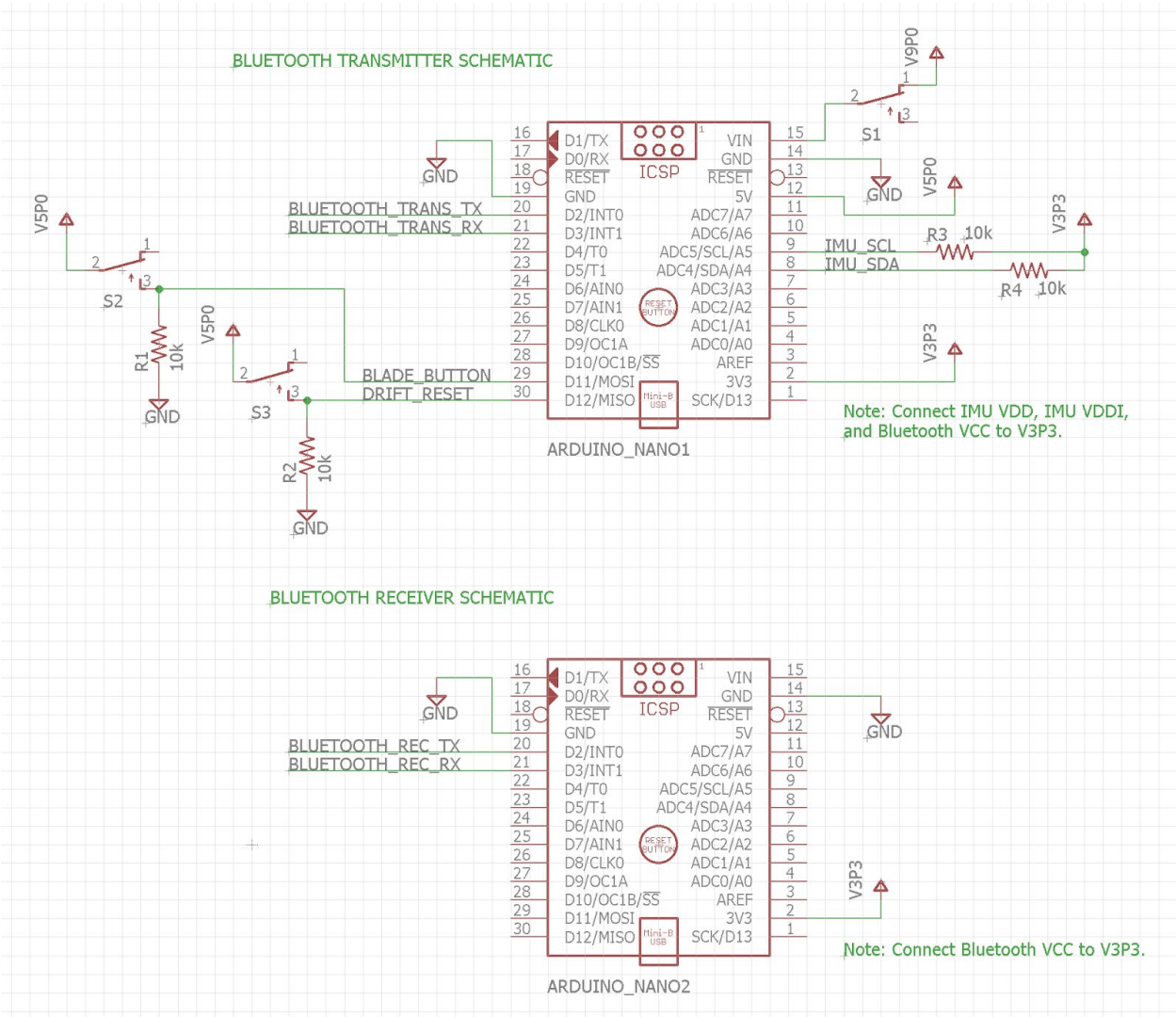
Overview

The first main part of our project was to wirelessly transmit data from the lightsaber handle and decode it on the FPGA. Specifically, the data that was transmitted was all 3 axes of the gyroscope and accelerometer data from the IMU (MPU9250) and the states of the blade button and the gyroscope drift reset button. The data was encoded by an Arduino Nano, sent out via a Bluetooth SMiRF Silver transmitter module, received by a Bluetooth SMiRF Silver receiver module, and then wired directly into the FPGA. The data was then decoded on the FPGA and filtered before being sent to the necessary video and gameplay modules.

Block Diagram



Schematic



Serial Protocol

Serial Port Profile (SPP)

The serial protocol used to communicate via bluetooth was SPP with zero parity bits. In other words, each packet in the protocol contains exactly 10 bits: 1 start bit (logic low), 8 data bits (LSB to MSB), and 1 stop bit (logic high). For this project, the communication rate used was 9600 baud.

Data Representation

For the IMU that we used (MPU9250), the accelerometer data ranged from -2g to +2g and the gyroscope data ranged from -250 deg/s to +250 deg/s. To retain some of the IMU's precision but not create too much overhead, I decided to use 8-bit values to represent the magnitude of each axis of the accelerometer and gyroscope data. To make it easier to handle negative values, I included an additional 9th bit as a sign bit, where 1 indicates a negative number and 0 indicates a positive number.

Serial Data Encoding / Decoding Protocol

I needed to transmit six 8-bit numbers with 1-bit signs each for the IMU data and 2 bits for the button data, which meant that a total of 56 bits of data needed to transmitted every cycle. Since it was important to know the order of the packets to decode the data, I needed to reserve the LSB of each packet to signal if the packet was the first packet (1) or not (0). Because of this, each packet could only transmit 7 bits of useful data at a time. Since there are a total of 56 bits of data to be transmitted, that meant that each cycle of data transmission would require 8 packets. To split up the 56 bits of data into its 8 packets, I used the following protocol:

Packet 1:	[first 7 MSB of magnitude of x-axis accelerometer data]		1
Packet 2:	[first 7 MSB of magnitude of y-axis accelerometer data]		0
Packet 3:	[first 7 MSB of magnitude of z-axis accelerometer data]		0
Packet 4:	[first 7 MSB of magnitude of x-axis gyroscope data]		0
Packet 5:	[first 7 MSB of magnitude of y-axis gyroscope data]		0
Packet 6:	[first 7 MSB of magnitude of z-axis gyroscope data]		0
Packet 7:	[LSB of accelerometer x, y, z, gyro x, y, z, respectively]	[blade_button]	0
Packet 8:	[sign bit of accelerometer x, y, z, gyro x, y, z, respectively]	[drift_reset]	0

Verilog Modules

read_data

The *read_data* module takes in the serial receiver_data and then decodes and stores the raw data. Specifically, the module is composed of two FSMs. The first FSM keeps track of the current packet number that is being read by the FPGA. Once this FSM detects a 1 on the LSB bit of the packet to signal that it's the first packet, a second FSM is triggered to read in each bit of the packet. When all of the bits in a particular packet are read, the second FSM signals to the first FSM to store the each bit value in the corresponding variable as per the packet protocol defined in the Serial Protocol: *Serial Data Encoding / Decoding Protocol* section of this report. The outputs for this module are the raw data for the IMU and buttons.

sample_timer

The *sample_timer* module is used to signal when the FPGA should sample the value of a bit. The timer is signaled to start by the *read_data* module at the beginning of every packet. Then, the timer asserts the enable signal once every bit to signal the *read_data* module to read the value of the bit. After the stop bit of a packet, the timer stops running, i.e. the enable signal remains deasserted until the next start_timer signal from the *read_data* module.

debounce

The *debounce* module is used to debounce the raw button data, i.e. raw_blade_button and raw_drift_reset, from the bluetooth receiver. The debounced blade_button output signal is sent to the Gameplay part of the project (see Part 3: Gameplay), and the debounced drift_reset output signal is sent to the *process_data* module.

process_data

The *process_data* module is used to filter the raw accelerometer and gyroscope data with a moving average filter. To deal with the signed aspect of the data, the format of the data is converted into two's complement, i.e. signed form, from its original 8-bit magnitude with a 1-bit sign bit form. Once this is done, the data is averaged and the

filtered average values of all three axes of the accelerometer and gyroscope data are outputted. These outputs are then sent to the Video part of the project (see Part 2: Video). This module also uses the debounced drift_reset signal to calibrate the gyroscope. Every time the drift_reset button is pressed, the *process_data* module stores the current gyroscope data values for all three axes as the new 'zero' for the gyroscope. This 'zero' value is then used as an offset for all incoming raw gyroscope data.

Arduino

Two Arduino Nanos are used in this project. The first Arduino is used to configure the Bluetooth SMiRF Silver transmitter module as the master and provide the module with data to send, as per the packet protocol defined in the Serial Protocol: *Serial Data Encoding / Decoding Protocol* section of this report. The second Arduino is used to configure the Bluetooth SMiRF Silver receiver module as the slave. (Note: the slave Bluetooth module must be powered on before the master Bluetooth module in order for the modules to connect with each other.)

The Github project link that contains both the transmitter and receiver Arduino code can be found in the Appendix.

Logic Level Differences

The IMU, Bluetooth modules, and FPGA all operate on a 3.3V logic level; however, the Arduino Nano operates on a 5.0V logic level. At first, I considered using level-shifters between all logic levels, but this turned out to not be absolutely necessary.

The Arduino Nano accepts digital inputs as high for voltages greater than 3.0V. Since the IMU I2C lines were not expected to deviate very much from their specs of 3.3V logic high, the Nano still was able to properly read the IMU I2C input, so logic shifting was not needed there.

To transmit the data wirelessly, the Nano had to send 5.0V logic level serial signals to the Bluetooth transmitter module. This technically was over the maximum acceptable input of the Bluetooth module, but it was within 1.0V of the maximum allowed voltage so the module was still able to function. (If this project were to be further refined, level-shifting circuitry should be

added between the Nano and the Bluetooth transmitter to not unnecessarily stretch the capabilities of the Bluetooth module.)

On the receiving end, since the Bluetooth receiver was outputting the received serial signals at a 3.3V logic level, there was no problem with directly hardwiring these signals to the FPGA, which also operates at a 3.3V logic level.

Power

To power the transmitter circuitry in the lightsaber handle without a tether, a 9V battery is connected to the input of the Arduino's onboard voltage regulators. Since the transmitter circuitry draws a small amount of current (on the order of a couple hundred mA), no additional regulators or power circuitry was needed. In addition, the battery is connected to a SPST switch for the user to turn on/off the lightsaber handle's circuitry.

Since the receiver circuitry is stationary and only needs to sit on the desk, the receiver Arduino is powered via a USB cable connected to a laptop.

Challenges

The biggest challenge for me was understanding SPP and developing a data protocol to communicate all of the IMU and button data wirelessly. First, I worked on understanding the Bluetooth SPP protocol by trying to send data wirelessly from the Arduino to my laptop. It took a while to figure out the necessary configuration for this to work. Once that was up and running, I had to design a packet data protocol to ensure that all the data I wanted could be transmitted in as few packets as possible.

After I had come up with the packet data protocol, I had to first write up the transmitter Arduino code and make sure the data was transmitting correctly. Only after I got this working was I able to even start writing Verilog modules to interface with the received data as without successfully transmitting data, I would have no way to test my code or have enough knowledge to properly design my Verilog modules.

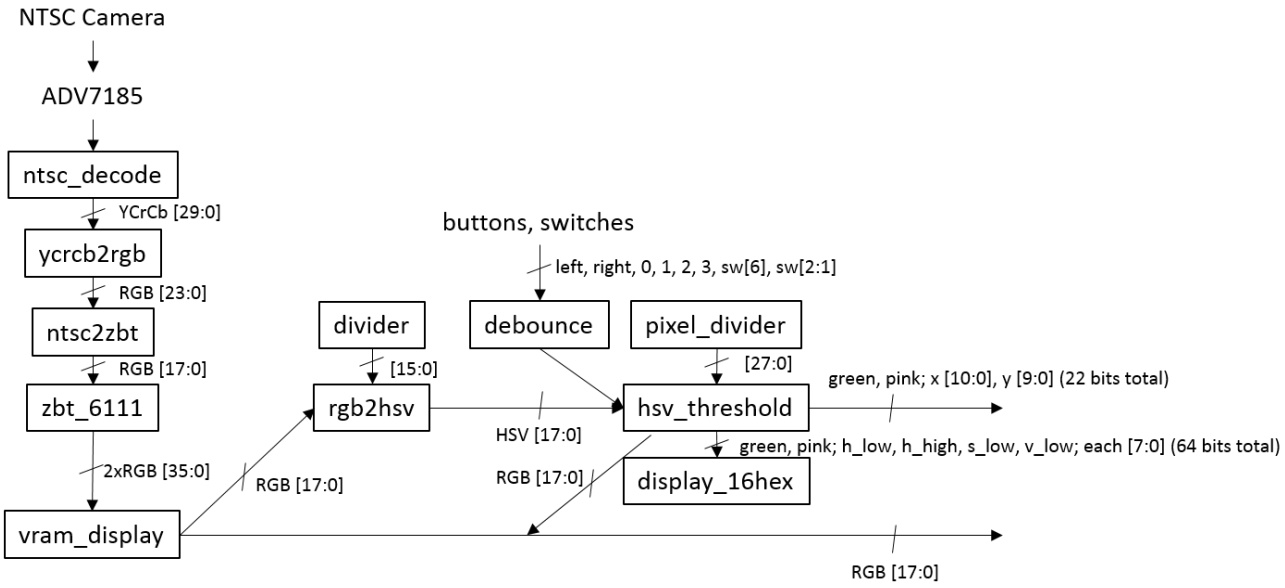
Once I got the Arduino code working, a lot of my time was spent refining the *read_data* module. My original design had only used one FSM to read in all 8 packets per cycle, but this ended up with greater than 90 states. So, I had to rethink my design to read in data and that's how I came up with the nested FSM design to greatly reduce the total number of states needed. After the high-level design of this module was completed, it still took many cycles of refinement to eliminate all of the bugs, especially timing-related bugs, to properly read in the receiver data.

Part 2: Video Processing (lcarter)

Overview

The central part of this project was to process NTSC video data to extract the position and size of the lightsaber hilt to the gameplay module. To do this, the video was read in, buffered through ZBT RAM, and processed via HSV thresholding and center-of-mass averaging to determine the centroids of the pink and green hilt ends. This data was then passed on to the gameplay module.

Block Diagram



Verilog Modules

ntsc_decode

This module was provided by the staff, and implements an FSM to read in NTSC data as 10:10:10 bit YCrCb data.

ycrcb2rgb

This module was generated by Xilinx IP Core to convert YCrCb data from the NTSC stream to RGB data for conversion to HSV and for display.

ntsc2zbt

This module was provided by the staff to buffer ntsc video through ZBT ram. However, the staff implementation buffered only 8 bits from the Y channel of the YCrCb video stream. Since we wanted to use color, I modified this code to buffer 18-bit RGB data instead. YCrCb was converted to RGB through a Xilinx IP Core module on the input, then ntsc2zbt was modified to store 2 pixels per address rather than 4. This allowed for the greater bit depth required for 18-bit RGB instead of 8-bit Y.

zbt_6111

This module was also unmodified from the staff implementation, but provided a pipelined ZBT ram writing module.

rgb2hsv

This module was also provided by the staff, and supplied with an IP Core divider module. This module converted the stored RGB data to HSV data for easier threhsholding. It is easier to consider thresholding over HSV than RGB, because human vision more closely approximates HSV than RGB. Furthermore, pink and green are further from one another in the HSV color space than the RGB color space, allowing for easier separation.

A pair of 16 bit dividers from Xilinx IP Core was required for each instance of this module to perform the linear transformation.

hsv_threshold

This module was the core of the video processing, taking in a stream of HSV data and thresholding it versus preset thresholds to detect green and pink values indicating that the pixel is part of the end of the lightsaber hilt. This module also performed the frame-averaging, which did center-of-mass averaging on the selected pixels. Pink and green pixels positions were averaged in x & y over each frame to determine the center of the x and y pixels. In future iterations, this should be replaced with proper blob detection - the provided NTSC cameras turned out to be noisy enough that a relatively large number of random pixels were included in the thresholding, even after optimizing the thresholding constants. It was not anticipated that the provided NTSC camera would be so noisy, so we figured simple center-of-mass averaging would be sufficient until testing it. We had further difficulties filtering for pink values especially, since the provided NTSC camera had quite low saturation, which made the pixel values for skin and the red end of the lightsaber hilt very similar, even when viewed in the HSV color space. This added a significant amount of noise if the Jedi user did not wear the proper hooded brown/black cloak.

In particular, the thresholding was performed with **min_hue**, **max_hue**, **min_val**, and **min_saturation**. These 4 values allowed a relatively intuitive control over thresholding for pink and green pixels. Furthermore, Switch 6 was employed to provide an overlay of which pixels were being selected, along with an overlay of the x & y values selected as the average. This helped significantly for debugging dumb mistakes. For example, when writing the averaging, I initially forgot to reset x_sum and y_sum on every frame refresh, which was revealed by the selected x and y values increasing on every frame refresh, even when only a few pixels met the thresholding criteria.

These 4 thresholding values were applied as follows for green:

```
assign green_detected = (in_frame &&
   cropped_hsv[23:16] > green_h_low &&
   cropped_hsv[23:16] < green_h_high &&
   cropped_hsv[15:8] > green_s_low &&
   cropped_hsv[7:0] > green_v_low);
```

And for pink:

```
assign pink_detected = (in_frame &&
   (cropped_hsv[23:16] < pink_h_low ||
    cropped_hsv[23:16] > pink_h_high) &&
   cropped_hsv[15:8] > pink_s_low &&
   cropped_hsv[7:0] > pink_v_low);
```

In particular, note that only in_frame pixels are allowed (in_frame was set by a simple cropping function). Also note that since pink is in the range of the 0/360 degree mark on the hue circle, values were OR'd between two extrema to include the 0/360 rollover.

Originally, I had intended to put a 3-5 frame low-pass FIR filter on the centroids to minimize the effects of noise. However, my implementation of this had bugs that I could not resolve in time, and furthermore the extreme amount of noise proved too much to be effectively filtered out anyway. In future iterations, enabling the FIR filter based on the accelerometer and gyroscope data would improve stability without sacrificing latency. In addition, blob detection would improve robustness to input noise.

A pair of 28-bit dividers was also required for each instance of this module to divide x_sum and y_sum by the number of pixels selected. This in particular added a significant amount of utilization and compile time to our project.

debounce

Debounce was used again in the video processing block, this time to debounce button input to adjust thresholding on the fly. Buttons 0-3, left/right, and switches 1 and 2 were debounced to provide control over the HSV thresholding. Switch 6 was used to select an overlay which showed selected pixels for green and pink thresholding, as well as the calculated centroids for the pink and green regions. This was invaluable for quick iteration. A small 20-bit clock divider was also set up to generate a ~60 Hz clock, where the values were incremented on the positive edge of the clock divider's MSB. This allowed the thresholding values to be tuned at a quick but manageable rate.

display_16hex

This module was used to display the threshold select values. Displayed on the left 8 hex digits were the values for the pink thresholding, while green thresholding values were displayed on the right 8 hex digits. This module was incredibly helpful for setting constants, since constants could be iterated in real-time rather than waiting for the 6-minute compile time of our final code. Since values for thresholding were displayed on the hex display, we could also copy them down and feed them into our program as initialization values for the thresholding registers.

divider & pixel_divider

These two modules were dividers generated through Xilinx IP core, and were 16-bit and 28-bit dividers, respectively for rgb2hsv and hsv_threshold. The especially-large divider was required for hsv_threshold for the corner case when every one of the pixels was selected, leading to an x_sum of over 2^27. Future iterations including blob detection rather than center-of-mass averaging could potentially remove this module, depending on the implementation (eg, skeletonization would not require a large divider, but simple connected-component thresholding on center-of-mass averaging would still require a large divider for the corner case of every pixel selected). These dividers necessitated a large delay in pixel throughput, so all hsv-corrected pixels were 22 cycles behind the RGB display. The effect of this was to right-shift all of the overlay data and (x,y) coordinates of the centroids by 18 pixels. Though the x_sum and y_sum dividers were larger, and thus had a larger delay (30 clock cycles, according to the Xilinx IP core spec), this delay was unimportant because it happened during the NTSC sync period, so it produced effectively no real-world latency.

vram_display

This code was also part of the staff-provided code, but modified for color. This code reads pixels from ZBT ram to display on the VGA monitor. It was originally written to display 4-pixel packed ZBT addresses as Y-only data, but I modified it to display 2-pixel packed ZBT addresses as 18-bit RGB data.

Coordination in overall "finallabkit.v" module

Overall, modules were hooked together very similarly to the staff implementation of NTSC video display, with the addition of hsv_threshold and the corrections appropriate to turn the signals from Y-only into RGB-enabled. Furthermore, to provide quicker iterations on hsv_threshold, buttons and the fluorescent hex display were hooked in to allow on-the-fly adjustment of HSV thresholding values. Switch 6 was also hooked in to toggle overlay of the pixels selected in the thresholding module, as well as the x,y coordinates selected by the averaging in the thresholding module.

Testing

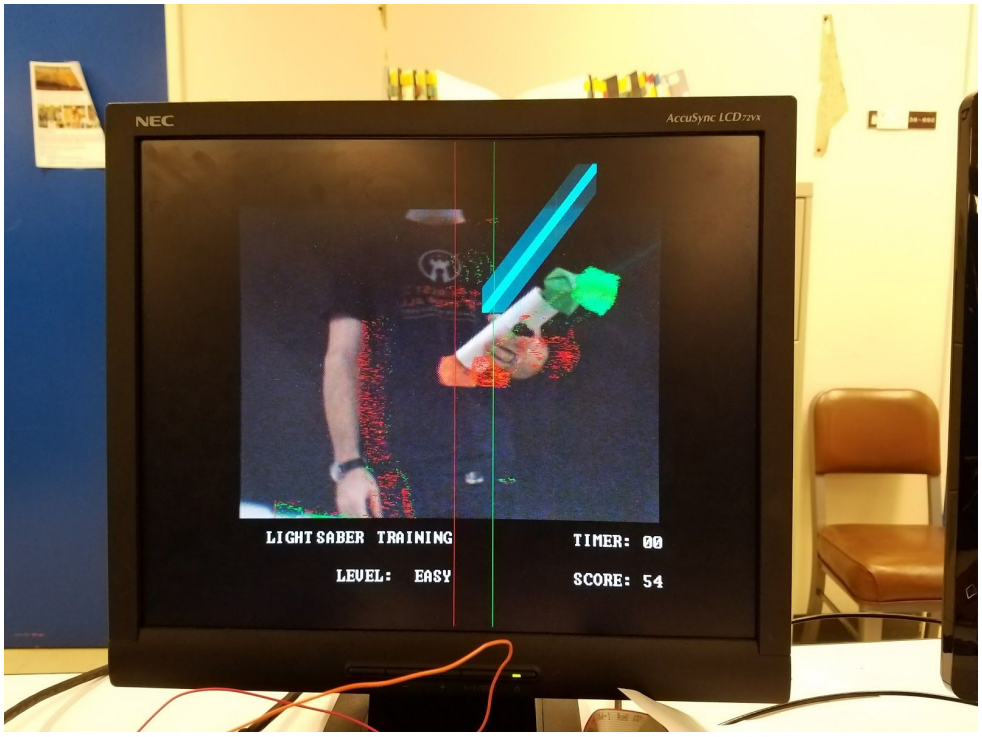
As with the gameplay module that follows, most of the testing and calibration was able to be performed visually. As previously mentioned, a small user interface was implemented with the switches, 16 digit fluorescent hex display, and buttons to allow users to tune the thresholding constants when switching cameras, lighting conditions, or if the module was behaving suboptimally otherwise. The values could be read off of the hex display and set as the initialization value for future iterations to prevent recalibration on every new firmware flash.

Challenges

The biggest challenge for me for this was modifying the ntsc code to work with color. Though I understood the concept, it was difficult to test changes incrementally because there was no nice intermediary between "working b&w" and "working color". One interesting mistake I made was to originally synthesize an 8-bit input YCrCb to RGB converter, while we had 10-bit YCrCb input. The effect of this was to chop off the two MSB's of each of the 3 channels, which led to a very psychedelic display:

After I had succeeded with the color NTSC conversion, the noise of the camera proved much higher than anticipated. This manifested as extremely noisy position estimates, even after careful thresholding. Setting everything up against a black backdrop and wearing black clothing and gloves helped significantly. Pink in particular was difficult, because the camera was extremely low-saturation. This manifested in skin being included in the pink position estimates. Even with blob detection fully working, this would have been a very difficult problem to tackle without simply wearing gloves.

Figure illustrating noise, as well as low levels of saturation. The pixel offset is due to the latency on the dividers.



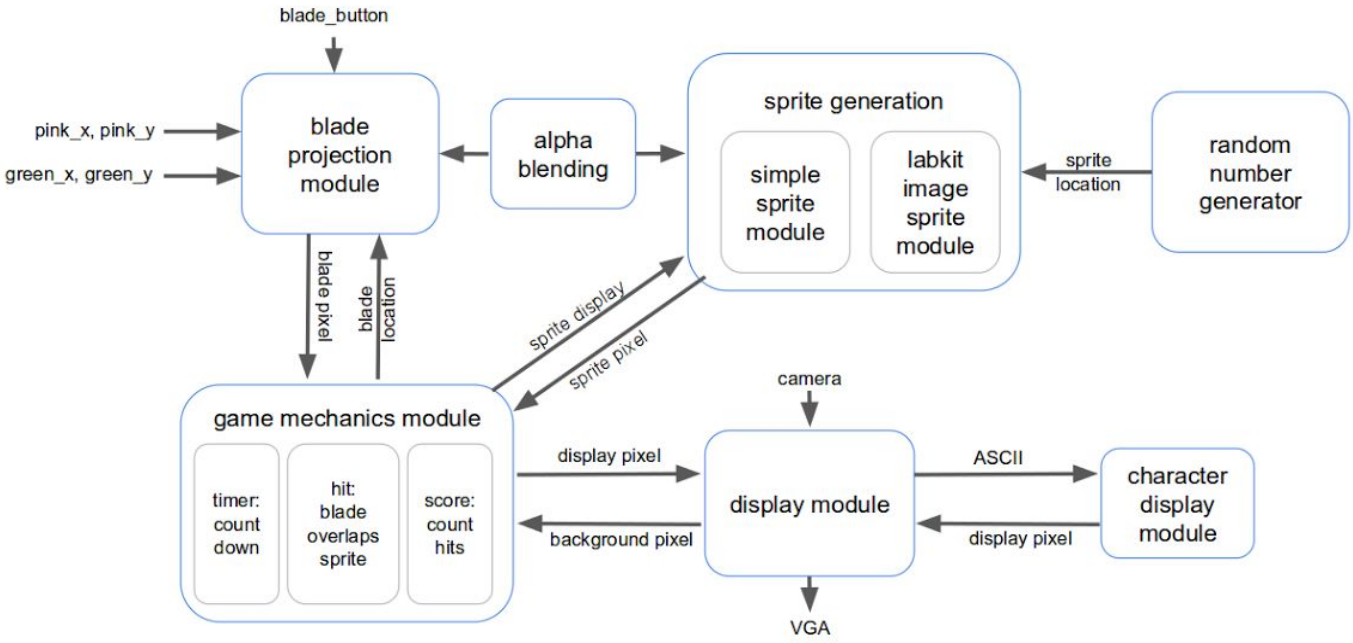
Even with a black backdrop and tuned constants, due to the low saturation, skin is included in the thresholding for pink, and some white edges are included in the thresholding for green.

Part 3: Gameplay (lolzhang)

Overview

The final part of the project was to take the sensor and image data and integrate the information into gameplay. The (x,y) coordinates of the handle were used to calculate the blade projection using dot products, and data from the button was used to extend and retract the blade as well as change its color. Sprites were generated with different properties based on the level selected by a switch on the labkit. On the “easy” level, the sprite would stay in place until the player hit it with the lightsaber, while on the “6.111” level, the sprite would take the form of a labkit and change position every second. The game mechanics also included a thirty second countdown timer and a score tracker that counted how many sprites had been hit; both the timer and score were displayed on screen.

Block Diagram



Verilog Modules

blade_block

The *blade_block* module displays the blade projection and takes hcount, vcount, (x,y) coordinates of the top and bottom of the handle, handle width, blade color, and background color from the camera as inputs. Its outputs are the 24-bit hex color of the pixel at (hcount, vcount) and an indicator of whether or not the pixel is occupied by the blade. I set the pixel location of the bottom of the blade to be equal to the top of the handle and then calculated the pixel location of the top of the blade by extending the handle vector to be three times the length of the handle. I then determined the area to be occupied by the lightsaber blade projection by taking the dot product of (hcount, vcount) and handle vector. If the dot product is less than a constant (either determined by the width of the handle or hard-coded) and (hcount, vcount) was in the blade space, then occupied is set to 1 and the pixel is shaded the appropriate blade color. Additionally, to make the lightsaber blade seem more realistic, I set the pixel to the blade color alpha-blended to the background color if the dot product was less than twice the previous constant. Otherwise, occupied is set to 0 and the pixel color is set equal to the background color. However, with this method I ran into a problem where if the handle was perfectly horizontal or vertical, then the blade projection would disappear because of the dot product math. I fixed this issue by displaying a rectangle if the x-coordinates or y-coordinates of the blade top and bottom were within a certain margin of each other. This solution ensures that the blade never disappears, even if the handle is pointed directly at the camera (the blade becomes a small square projection). I also solved a problem with the blade wrapping around the screen by using signed variables and increasing the size of the registers for the blade and handle coordinates. Finally, I accounted for the 22 pixel offset due to the hsv-corrected pixels being 22 cycles behind the RGB display, by subtracting 22 from the x-coordinates of the blade handle. Boundary cases were not an issue because the camera image was in the middle of the screen surrounded by a border larger than 22 pixels.

sprite

The *sprite* module displays a circular sprite and takes hcount, vcount, an (x,y) position, sprite color, and background color from the camera as inputs. Its output is the 24-bit hex color of the pixel at (hcount, vcount) and an indicator of whether or not the pixel is occupied by the sprite. I check if (hcount, vcount) is in the circle with center (x,y) with a constant radius and set occupied to 1 and the pixel to the sprite color if so. If not, then I check again with double the previous radius and set occupied to 1 and the pixel to the sprite color alpha-blended with the background color. Otherwise, occupied is set to 0 and the pixel is set equal to the background color.

alpha_blend

The alpha_blend module makes objects look transparent by blending colors and takes the 24-bit object color and background color as inputs. Its output is the 24-bit hex color of the alpha-blended pixel. The formula used is as follows, with $0 < \alpha < 1$:

$$R(\text{blended}) = R(\text{object}) * \alpha + R(\text{background}) * (1- \alpha)$$
$$G(\text{blended}) = G(\text{object}) * \alpha + G(\text{background}) * (1- \alpha)$$
$$B(\text{blended}) = B(\text{object}) * \alpha + B(\text{background}) * (1- \alpha)$$

However, since Verilog does not have built-in dividers, I used multiplication and right-shifting to achieve the equivalent of division.

picture_blob

The *picture_blob* module displays an image of the iconic 6.111 labkit and takes hcount, vcount, an (x,y) position, and background color as inputs. Its output is the 24-bit hex color of the pixel at (hcount, vcount) and an indicator of whether or not the pixel is occupied by the image. I displayed the .JPG image of the labkit by following the instructions outlined on the 6.111 tools webpage. I first converted the .JPG image to a 256 color .BMP file using GIMP and then modified the MATLAB script sample to generate .COE files of the red, green, and blue color maps, as well as a mapping from the image pixel address to the color palette. I then added four Block ROMs to the project using the generated .COE files as sources. The pixel address to color palette Block ROM had a width of 8 and a depth of 7700 (since the image was 100 pixels by 77 pixels), while each color map Block ROM had a width of 8 and a depth of 256 (since I used a 256 color palette). In the picture_blob module, I calculated the ROM address of

the (hcount, vcount) pixel and read the location to use in the color map to create the 8 bits of red, 8 bits of green, and 8 bits of blue. I then assigned occupied to 1 and the pixel color to the color mapping if (hcount, vcount) was in the image space. Otherwise, I assigned occupied to 0 and the pixel color to the background color.

random_number

The *random_number* module generates a pseudorandom number and takes a seed value, new value indicator, and offset indicator as inputs. Its output is a 10-bit pseudorandom number. On reset, the number is set to the seed value. At each clock cycle, the next number is generated using a linear feedback shift register as shown below:

num_new[9] = 0
num_new[8] = num_old[7]
num_new[7] = num_old[6]
num_new[6] = num_old[5]
num_new[5] = num_old[4]
num_new[4] = num_old[3]
num_new[3] = num_old[2]
num_new[2] = num_old[8] XOR num_old[7]
num_new[1] = 0
num_new[0] = 0

This generates a pseudorandom binary sequence, which exhibits statistical behavior similar to a truly-random sequence. When the new value indicator is asserted, then the output number is assigned to the current random number; otherwise, the output number stays the same. When the offset indicator is asserted, then an offset constant is added to the calculated random number. The reason I added this offset option is because the *random_number* module is used to generate random locations for the sprites, and I wanted to minimize the probability that the sprite would generate onto the blade, which would add unintended points to the player's score.

bcd_ascii

The *bcd_ascii* module calculates the ASCII code of a decimal number from its corresponding binary input. Its output is the decimal 1's place and decimal 10's place in

ASCII. The module first converts the binary number to binary coded decimal (BCD) using the double-dabble method, also known as the shift-and-add-3 algorithm. Because *bcd_ascii* is used exclusively for outputting the time and score on screen, I decided to convert the outputs to ASCII within the module by adding an offset of 48 to each decimal place.

char_string_display

The *char_string_display* module displays an ASCII encoded character string in a video window at some specified (x,y) pixel location. The code is unmodified from the staff implementation.

one_hz

The *one_hz* module is asserted high once per second and takes a 65MHz clock and a start indicator as inputs. Its output is a 1Hz enable signal. At every clock cycle, a counter is incremented by 1, and when the counter reaches 65,000,000 then the output is set to 1 and the counter is reset to 0. Otherwise, the output is set to 0.

timer

The *timer* module counts down from a given number and takes a start time parameter, start indicator, and one hertz enable signal as inputs. Its outputs are a countdown number and a time expired indicator. When start is signaled, then the countdown number is set to the start time and the time expired indicator is set to 0. At every one hertz enable signal, the countdown number is decremented by 1. When the countdown reaches zero, then the timer stops counting and the time expired indicator is set to 1.

gameplay

The *gameplay* module contains the game mechanics and takes hcount, vcount, (x,y) coordinates of the top and bottom of the lightsaber handle, the level switch (switch[0] on the labkit), and the background color from the camera as inputs. Its outputs are the 24-bit hex color of the pixel at (hcount, vcount), the time left in countdown, and the player score. The module stores the state of the blade (on/off and color), which changes when the button on the handle is pressed. The blade cycles through four states as the button is pressed: off, blue, magenta, and green. Additionally, the player can change the level

of gameplay by toggling switch[0] on the labkit. On the “easy” level, the *sprite* module is used, and the sprite changes position randomly only if it is hit by the lightsaber blade. On the “6.111” level, the *picture_blob* module is used, and the sprite changes position randomly every second, as well as when it is hit by the blade. The player is able to immediately change levels in the middle of gameplay. To decrease the probability of the sprite generating on top of the blade, an offset is added to its x-coordinate every other location change so that the sprite appears on the other side of the screen. A hit is determined by tracking whether any pixel on screen is occupied by both the blade and the sprite. If a hit occurs, then a new (x,y) value is randomly generated for the sprite, and the player’s score is incremented by 1. However, the player can only hit sprites if the timer has not expired. Once the timer expires, then the sprites disappear from the screen and no additional points can be scored, although the lightsaber blade projection still remains on screen for the player to practice with.

Coordination in overall “finallabkit.v” module

Integration in the overall labkit module proved to be more difficult than expected. All three parts of the project had been using the labkit hex displays to display data, and in the end, we ended up displaying the thresholding values for quick adjustments. Additionally, the resolution of the camera image was smaller than the resolution of the screen, so I had to adjust the placements of the timer, score, and level strings to be more aesthetically pleasing. I chose to allow the sprites and lightsaber blade projection to extend outside of the camera image rectangle and into the black borders. This forces the player to step closer to the camera to make the blade longer if they want to hit the sprites that are near the edge of the screen and results in more active gameplay.

Testing

Because almost all of the gameplay is displayed on screen, I was able to test the majority of my code visually. To test the blade projection, I set the bottom of the handle to a constant point near the center of the screen and the top of the handle to be a point controlled by the up, down, left, and right buttons on the labkit. This allowed me see how the blade projection responded to changes in the handle angle and length. I was able to incrementally add each component of

gameplay so that the user could play the full game by using buttons on the labkit before integrating the other parts of the project. Additionally, I was able to test the game mechanics by displaying the timer and score on the hex digits on the labkit before displaying them on screen.

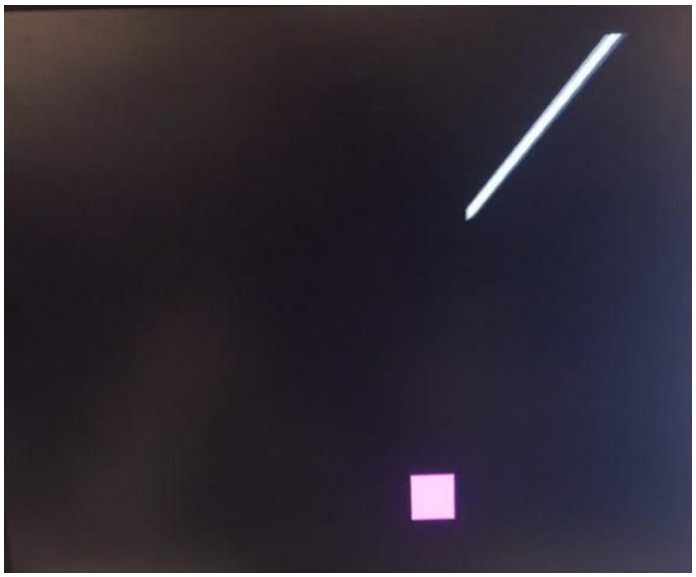


Figure illustrating commitment level project testing with solid white lightsaber blade and solid square magenta sprite.

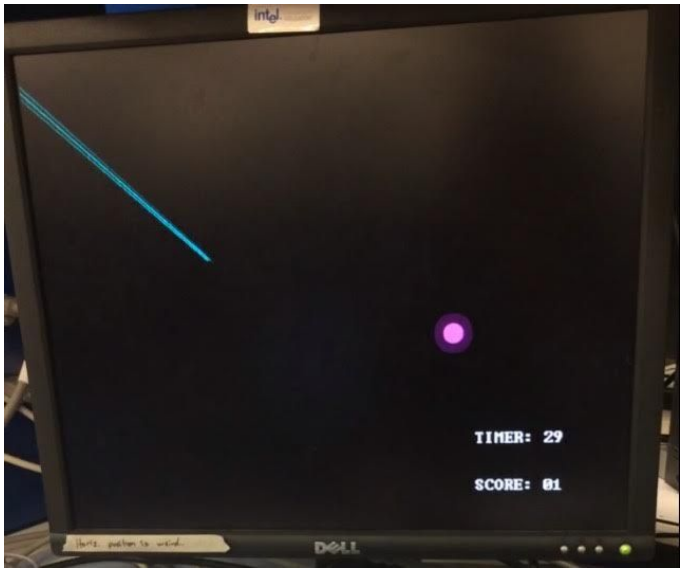


Figure illustrating goal level testing with alpha blended lightsaber and sprite, as well as timer and score displayed on screen.

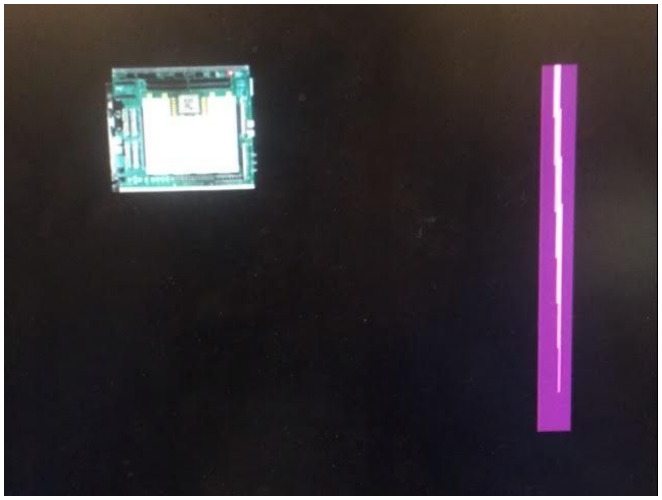


Figure illustrating stretch goal level testing with .JPG image sprite and different color lightsaber.

Challenges

There were a few main challenges I faced with the gameplay portion of the project. The first challenge was determining the best way to display the lightsaber blade. I researched a few algorithms for generating graphics, including Bresenham's line algorithm and Xiaolin Wu's line algorithm. However, both of these algorithms are incremental, and I would have had to do all calculations during the blanking period. Additionally, it would have been difficult to make the top and bottom ends of the blade perpendicular to the handle angle. Thus, I decided to use my dot product method, which could be calculated quickly for each pixel and naturally made the top and bottom ends of the blade pointed. While this method had its own drawbacks, including the problem with vertical and horizontal dot products mentioned previously, I believe that it was the optimal choice after the fixes I made. I think that although I spent a lot of time researching the benefits and drawbacks of the different algorithms, I ultimately saved time by not trying to immediately implement the first one I stumbled upon and also learned more theory. I would definitely recommend doing substantial research and planning before jumping into coding.

Another challenge I faced involved timing. I originally had many multiplications in my blade projection and sprite calculations, which did not meeting timing limits. Because of this, there seemed to be a lot of “noise” in my graphics when I was testing. I fixed this problem by pipelining some of my multiplications and decreasing the sizes of my registers. These problems reinforced the idea that Verilog is based in hardware, and must satisfy the restrictions imposed by hardware. In the future, I would advise others to keep in mind the timing requirements and not add unnecessary bits to registers to prevent this problem.

Finally, the last challenge in this project revolved around integration of the gameplay and video processing. I ran into a problem where the camera image would override the gameplay projections instead of the other way around, which led me to find a bug in my gameplay logic that I couldn’t catch in testing. Additionally, there was a problem where Landon and I used switch[0] to display different things, which led to a display of red and blue bars instead of the camera image when the player was on “6.111” level. This could have been avoided with more careful communication while integrating, which I would advise for everyone, and not just during final projects.

Conclusion

We were able to achieve all of the goals we had initially set for this project. For the sensor data part, we successfully transmitted all data via Bluetooth and filtered this data for use in other modules as well as incorporated gyroscope drift compensation. For the video part, we achieved reasonably accurate position sensing of the two ends of the lightsaber hilt with no noticeable lag. For the gameplay part, we had an alpha-blended lightsaber blade projection and dynamic alpha-blended sprites as well as the player score, game level, and timer displayed on-screen.

Besides reaching our goals, we were also able to achieve some of our stretch goals. Although we didn't have time to extend our project to work with a physical quadcopter, we were able to change lightsaber colors via the wireless blade button, have preloaded images used as sprites, and had a completely untethered lightsaber handle. All video processing was also done in a timely manner so that there any lag was imperceptible to the human eye.

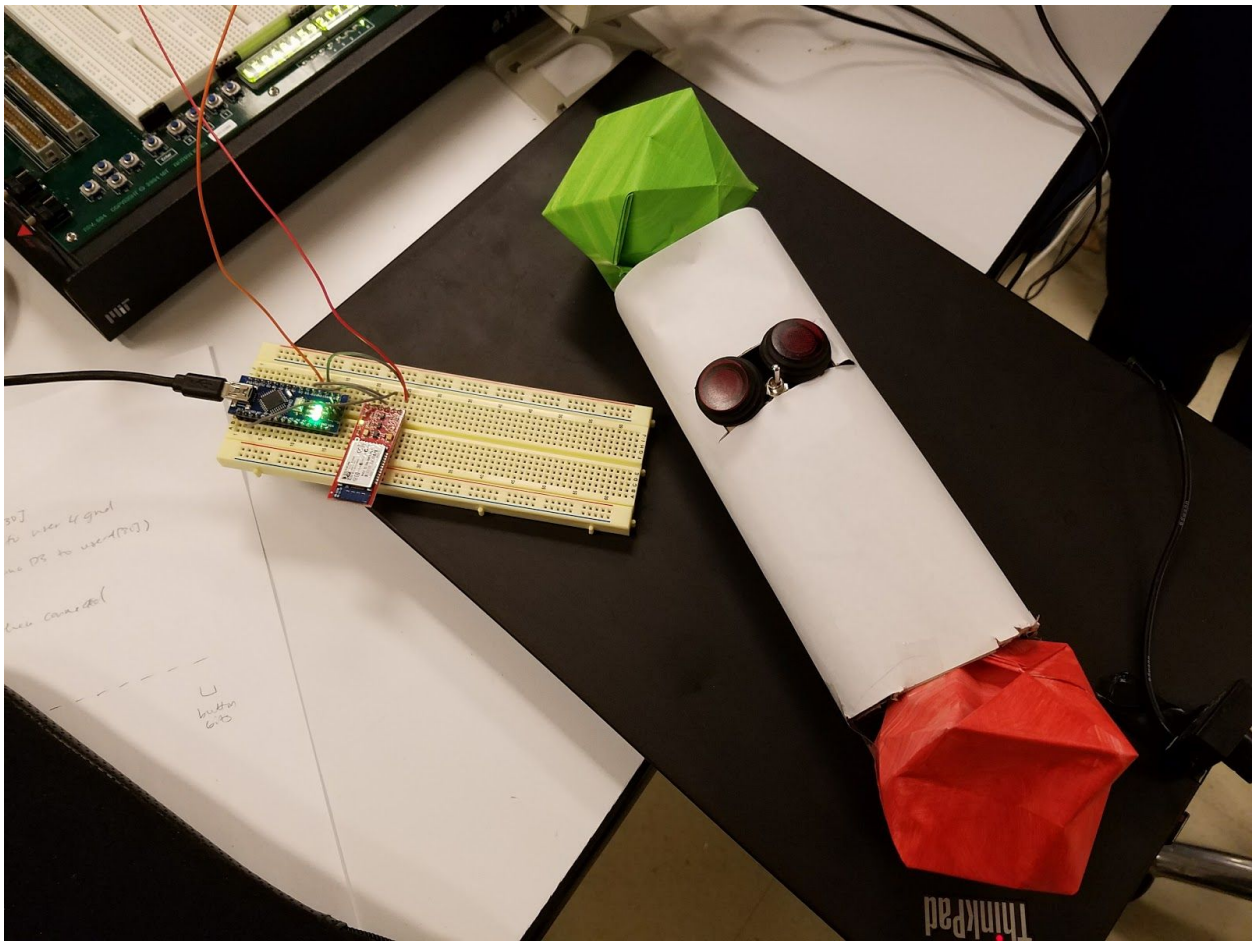
The material learned while working on this project has been quite rewarding. We would like to thank Professor Hom, Mitchell Gu, and the rest of the 6.111 staff for providing us with advice and guidance on this project.

Appendix

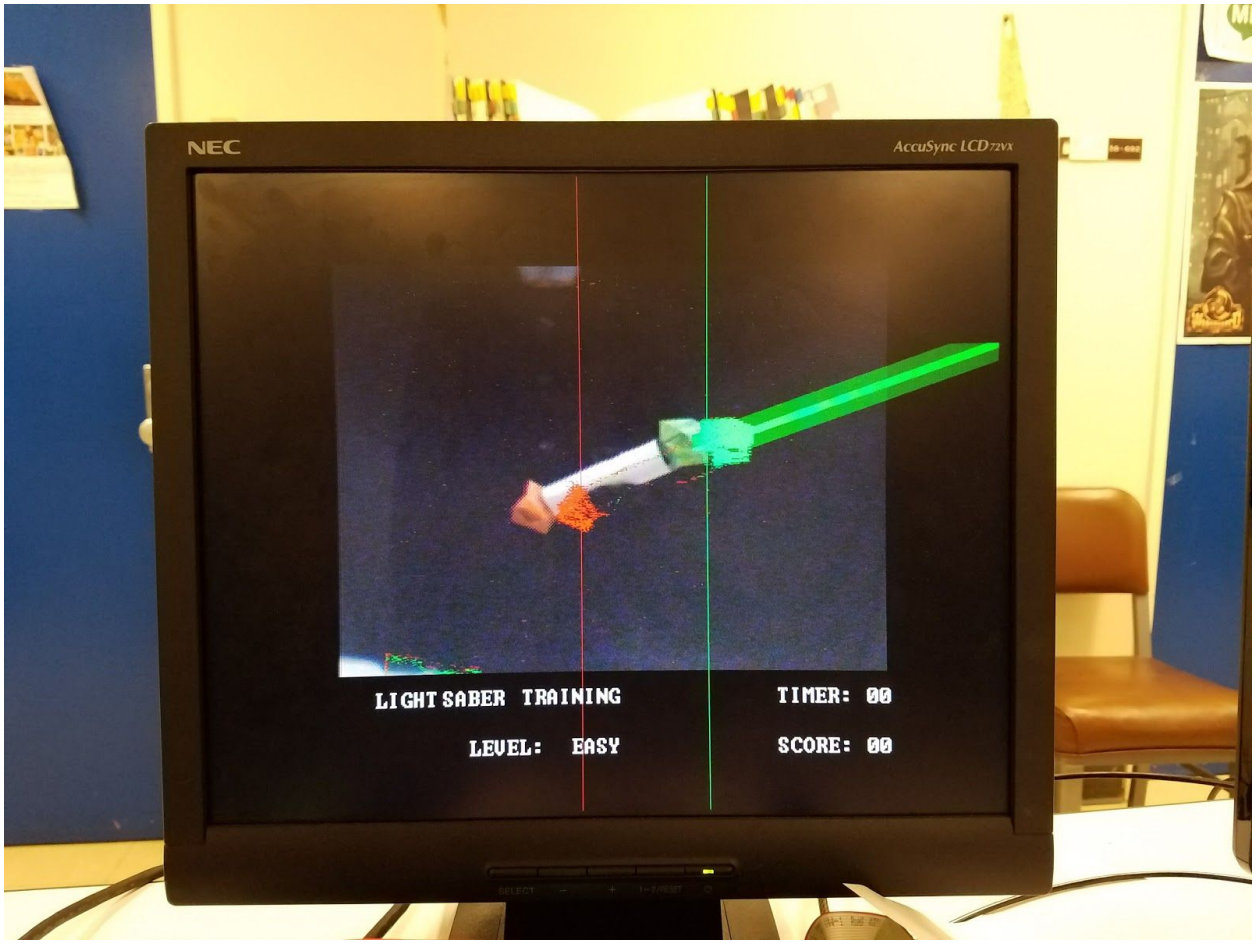
Github Link

Our Verilog and Arduino files can be found here: https://github.com/lycarter/6.111-finalproj/

Pictures



Final lightsaber hilt with buttons for accelerometer/gyroscope reset and blade color change.



Final user interface with thresholding overlay turned on (only x-values, no blob-center y-values). Note the pixel offset due to divider latency. This was manually corrected for in the gameplay module.