Digital shooting range

Emmanuel Azuh
Mubarik Mohamoud
6.111 Final project proposal

Abstract

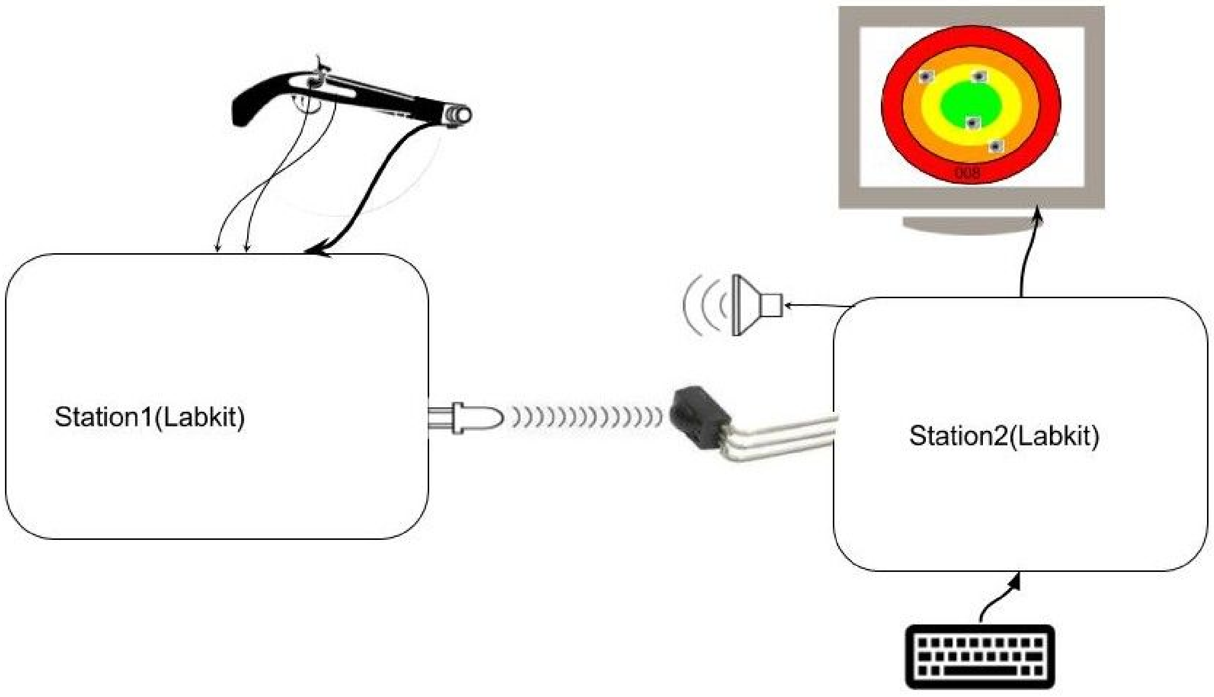
There were several 6.111 shooting game projects over the years including the Rifle Arcade game from 2014 and the Beat Gunner from 2009. Rifle Arcade and Beat Gunner are quite interesting and immersive games but they both have significant limitations in terms of range. Beat Gunner uses a photodiode as the tip of the gun to determine whether the bullet missed or hit the target based on the brightness of the incident point, the screen goes dark during the clock cycle following a shot except the target stays bright. For Rifle Arcade, they used a camera to detect the tip of the gun which is a small, blue ball then they used a gyroscope to determine the angular rotations of the gun to estimate the point of incident.

In Beat Gunner, there is a hit and there is a miss but nothing in between and the photodiode limits their range to about a meter. For Riffle Arcade, determining the tip of the gun using the camera can be problematic. The camera looks up a blob of a certain color in space, which works well only if the background was picked to avoid false positives. Otherwise, it would require more complex, resource intensive, multistep algorithms like background removal to distinguish the tip from other static objects that may look the same. Additionally the tip of the gun needs to be big for longer distance tracking.

Digital Shooting Range is similar to those projects in many ways but determines the point of incident of a shot in a simple and intuitive way, that allows long range tracking. Like Rifle Arcade, we are using a camera to determine where the player is aiming, but in this case the camera is the tip of the gun and we use it to find the target. In our case, the target is a VGA monitor displaying distinct patterns of colors, concentric circles.

This is intuitive because it is the shooter that's determining where the target is instead of the target determining where the shooter is. And it is simple because all we have to find is the center of the target with respect to the view of the camera, and then we simply determine our score based on the distance between the center of the camera view and the center of the target.

High level Overview



The above figure shows what the hardware and the software of our system looks like in high-level. We used two Labkit FPGAs to implement the project.

The first FPGA(station1) receives NTSC video from the camera on the tip of the gun. We use the video data to find the target, which is a vga monitor displaying concentric circles as shown to the top right of the figure. The gun also has two push button switches, which are wired to the to the FPGA, to indicate whether the shooter is taking a shot(trigger) or is done with his/her round(clear). Using image processing, we find the 2D position of the center of the target in the camera's view. If the center shows up, we transmit the position over IR when either trigger or clear is pushed.

The second FPGA(station2) receives the transmitted data via IR receiver and performs the game logic to display the target pattern, score, and the bullet marks. It also accepts keyboard input to allow the shooter customize the screen with her/his name.

Design

In deciding our final project, we wanted to work on a mixture of skills we have acquired from labs and new topics. Our skills from lab5B and lab3 would help us with the transmitting and receiving IR as well display modules. We would also explore new ideas like image processing along the way.

When we came up with Digital Shooting Range, we knew we wanted to optimize for mobility and accuracy in order to mimic an actual shooting range as closely as possible. Putting the camera on the gun was "killing two birds with one stone". With the camera as the tip of the gun, not only can we move around and shoot from wide range of angles and distances, but also we can also determine where we hit with pixel accuracy.

Implementation Diagram

Figure 1

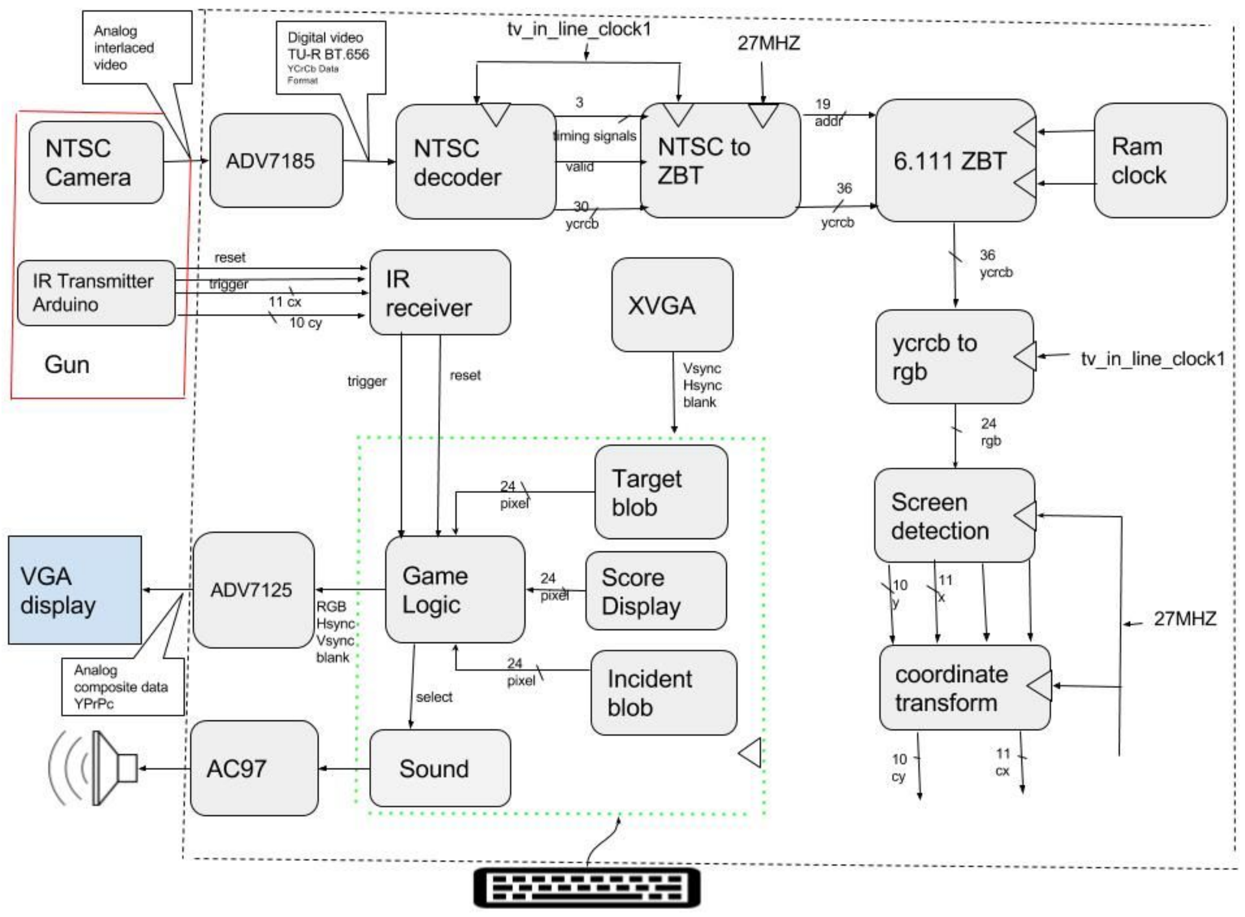
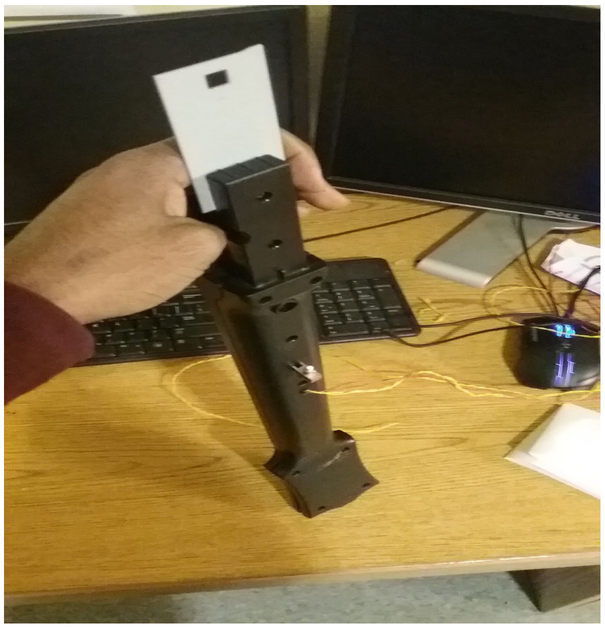
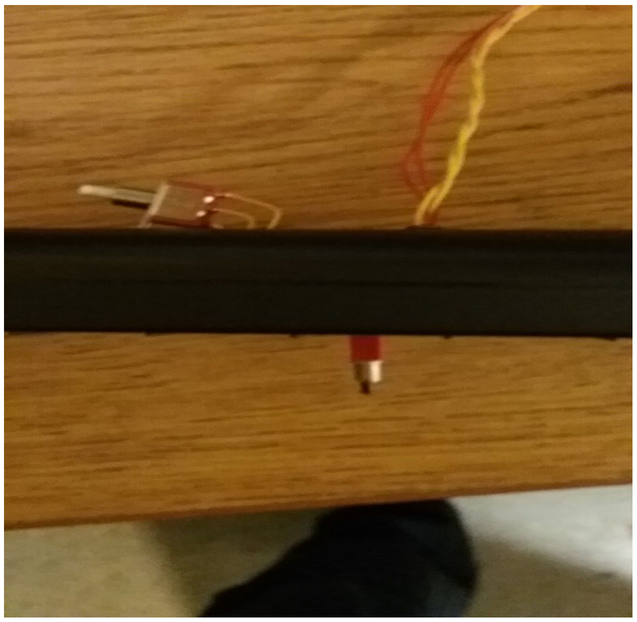


Figure 1 shows the high level block diagram of our implementation with all the major modules and their connections shown. For more explanation of each module, keep reading.

Gun(Mubarik)

To construct something that resembles the look and the feel of a gun, we removed the body of a Guitar Hero instrument, poked a couple of holes to fit the push-button switches for trigger, and clear switches, and finally attached a plate with screw holes at the tip to hold the camera. We placed the trigger switch in the middle and pointing forward. This orientation allows for one to push back with one’s index finger the same way the trigger of an actual gun would work. The clear switch points up just behind the trigger switch so you can easily push it with your thumb without moving your hand.

NTSC Camera(Mubarik)

The NTSC camera, which was available in the lab, provides us with analog video data in the YUV (ycrcb) domain as well as synchronization signals. We transformed this information into domains we were more familiar with (thus making the image easier to manipulate) like RGB and HSV. We then used the images in those domains to detect the target. More about the NTSC camera.

NTSC Decoder(Mubarik)

The NTSC Decoder module is the interface with the NSTC camera that extracts the video data and the sync signals from the video jack input of the labkit. This module had already been implemented by Javier Castro as part of the NTSC sample verilog available on the 6.111 website.

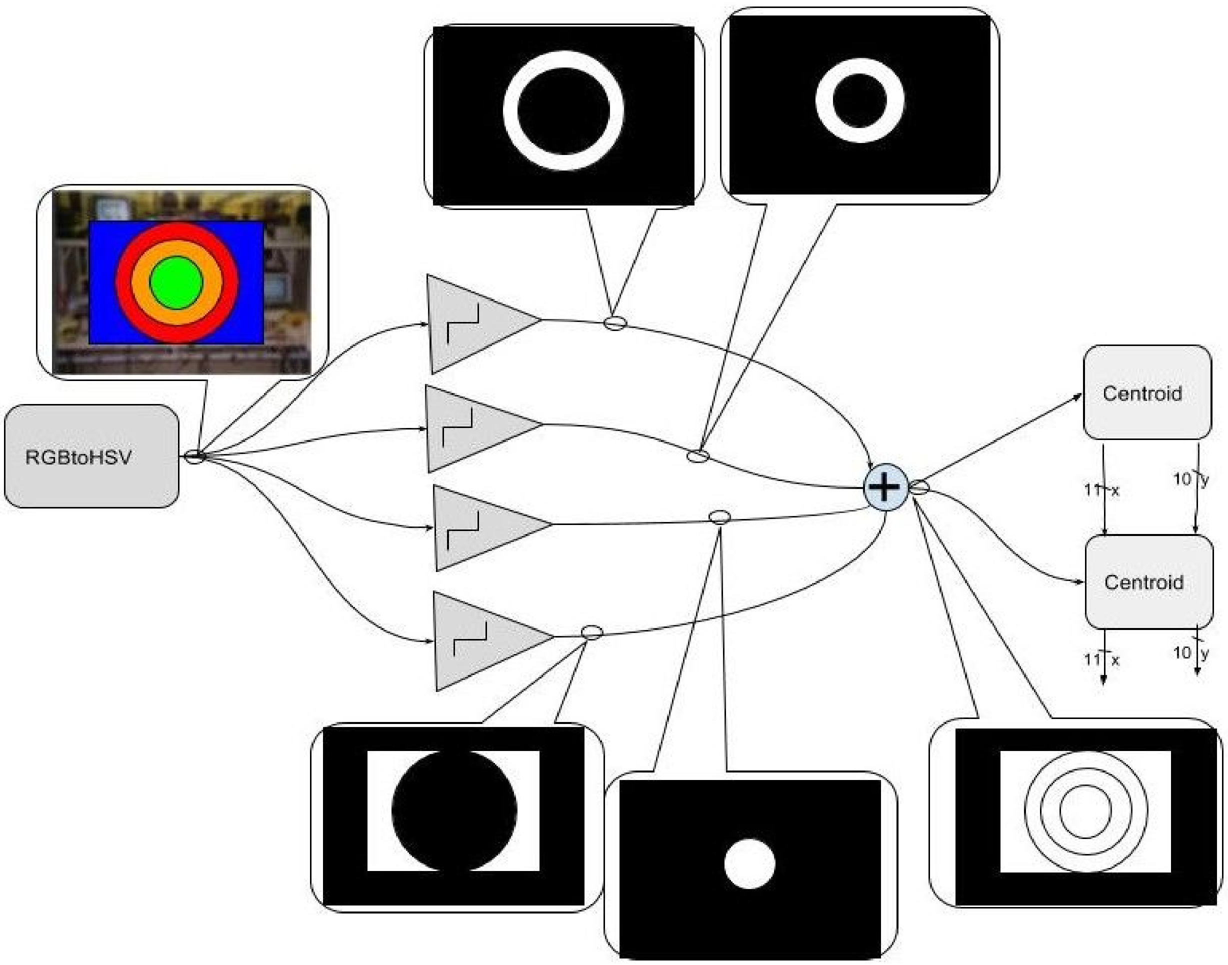
NTSC To ZBT(Mubarik)

This module provides an interface for writing to the Labkit ZBT RAM. It takes video data from the NTSC decoder and stores it on the Labkit’s ZBT RAM, from which we can read the stored data for our object detection modules. Most of this module was also available here but we modified it to store RGB image instead of the grayscale image.

ZBT RAM Reader(Mubarik)

This module which is also available as part of the NTSC sample verilog was used to read the video data and signals from the ZBT memory.

Target Centroid Detection(Mubarik)



We performed multiple parallel thresholding operations to select the pattern of our target. The resulting image was an 8-bit grayscale image, which we then performed centroid detection on. Centroid detection operation involves averaging the 2d coordinate locations of the white colors as the following formula shows.

$x_{center} = (\sum_x x)/(n)$, *n => the number of white pixels*

$y_{center} = (\sum_y y)/(n)$, *n => the number of white pixels*

Our threshold was sensitive to noise especially when our target shows up somewhere close to the top left of the frame and there's noise showing up some where close to the bottom right corner. If the changes in x and y introduced by the noise were significant, they dragged the center from the ideal location by a significant amount. Sometimes, the noise were significantly large blobs (including people working around).

To solve this problem, we made two assumptions. The first was that the target is significantly larger than the noise blobs, and the noise blobs were scattered around in space. Secondly, we assumed that noise further away from the target influenced the centroid a lot more than the closer ones. We then re-calculated the centroid using a rectangular region around the centroid from the previous frame. The formulae for re-calculating the centroid are as shown below:

$$x_{center} = (\sum_x x)/(n), (x_{center0} - W) \leq x \leq (x_{center0} + W)$$

$$y_{center0} = (\sum_y y)/(n), (y_{center0} - H) \leq x \leq (y_{center0} + H)$$

One needs to understand that calculating the centroid is not as easy as it seems from the above equations because one has check whether the pixels in the frame are white or black. There are two possible ways of calculating the columns and the rows.The first one is using the sync signals directly from the NTSC, and the second one is using hcount and vcount from the xvga module after reading from the ZBT memory. The NTSC decoder module provides *data_valid* signal which indicates when you have a full pixel(Note: The decoder has to wait for two pixels to get both the Cr and the Cb components of the YCrCb pixel ready).

The downside is that pipelining in this space can be tricky and complicated because you have to know how many clock cycles it takes to get a pixel ready. The decoder has 16-state FSM that transitions based on signals from the camera, and it is hard to see when those transitions happen. However, you may get away without pipelining as long as you keep your delays reasonably low. With delay of around 18 clock cycles, we got a small artifacts on the left side of the frame.

Coordinate Transform(Mubarik)

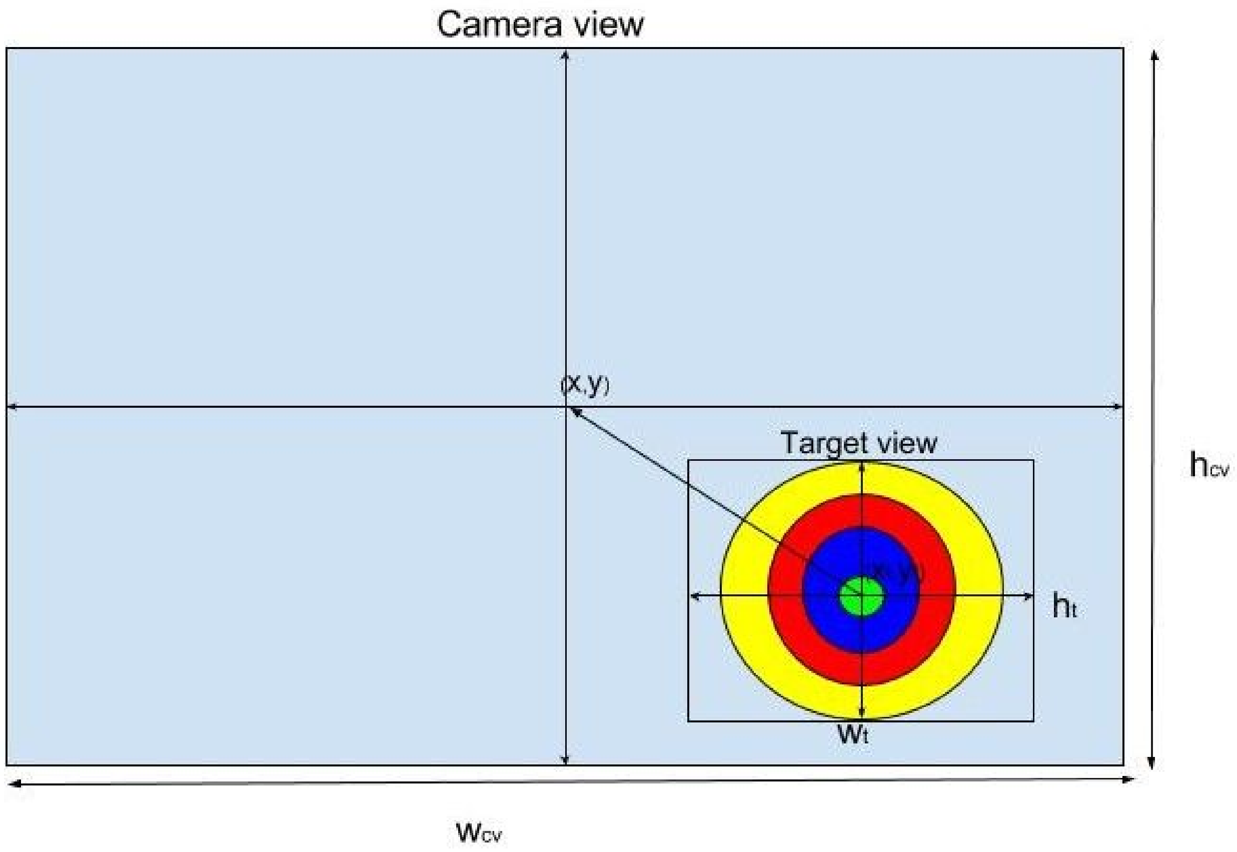
Calculating the center of the target wasn't going to be enough for us because the game would be too easy. With the centroid only, all you have to do is have the target show up on the camera frame and you have a hit. To make things more realistic, we wanted to have the center of the camera frame hitting the target in order to get points. The following equations transform the centroid we have calculated above from camera perspective to target prospective.

$$y = \frac{h_{cv}}{h_t}\left|y_t - \frac{h_{cv}}{2}\right| \qquad x = \frac{w_{cv}}{w_t}\left|x_t - \frac{w_{cv}}{2}\right|$$

h_{cv}: Height of the camera frame.

h_t : Height of the target with respect to the camera view.
w_{cv} : Width of the camera frame.
w_t : width of the target with respect to the camera view.
(x_t, y_t) : Center of the target.

Figure 2



Calculating the Height and Width of the Target(Mubarik)

To do the coordinate transform, we have to find the height and width of the target in the camera frame and by far was the most challenging part of the image processing. We tried couple of algorithms to find the height and the width most of them didn’t work to our liking.

The first algorithm involves keeping track of the roll of the gun using three-axis accelerometer. If the accelerometer gives us how much the camera is tilted, then we could slide two straight lines across the frame, one to fine the bottom and the top of the target, and the other to find the two sides. For each line, we take the first such a line that exceeds some number of white pixels and the last such a line to also exceed the same number of pixels. For example, when the camera has zero tilt, we slide a horizontal line and a vertical line across. The slopes of such lines are determined from the tilt. The problem with this algorithm is that the accelerometer drifts very quickly, and recalibration wasn't something we wanted to do because it restricts the mobility of the system.

The next algorithm we tried involves sliding small square across the frame and taking four of such squares that exceeds some number of white pixels inside them, and most likely to be the four corners. We were basically looking for combinations of x and y. For example, the square with the smallest x value would either be the top left corner or the bottom left corner and we would separate them based on their y-values. The problem with this algorithm was that we needed to assume that there will not be other random dense blobs showing up in the frame.

We tried to make this assumption true by trying dilation followed by erosion (opening), and erosion followed by dilation(closing). However, this didn't work to our liking; we were sometimes amplifying the noise and other times erasing most of the target blob. After all we think we gave up on this idea too early, but it was problematic still.

The algorithm that worked well enough was centroid detection. We basically split the frame at the centroid we calculated above, thresholded blue which was concentrated on the corners, and calculated the centroid for each region of the four quadrants. We again bounded each region based on the manhattan distance between the region centroid and the blob centroid and recalculated the centroids.

Image Processing Visuals(Mubarik)

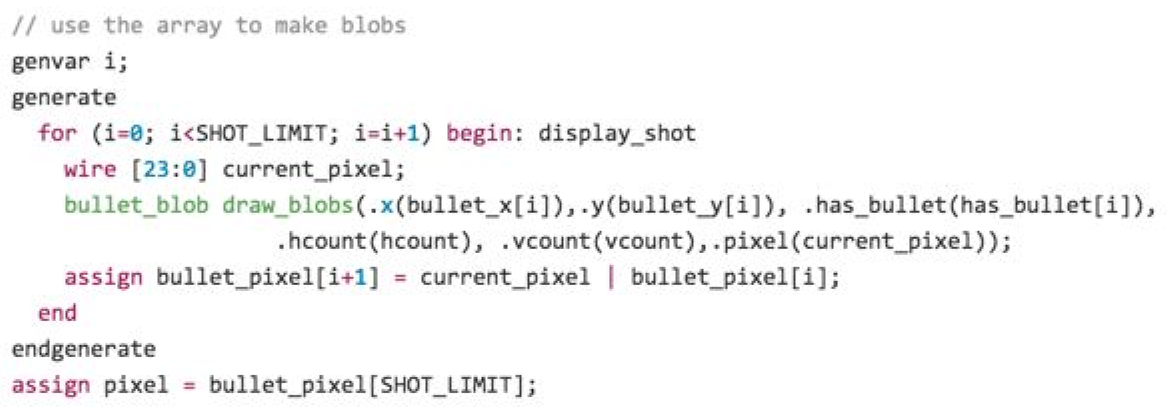
For someone who has gotten used to high-level and script languages, hardware description language can be painful to debug and visualize - no easy to use compilable print statements, and ModelSim can sometimes take long time to work and introduce a lot of problems that are difficult to detect (e.g. initializing arguments only works in simulation).

To make this less of a problem, We were displaying every version of the image on a VGA monitor as well as blobs on the points of interest(e.g. We were displaying small squares on the centroids) using the Labkit switches to select what to show. We also wrote a line drawing module to see how well our corner detection was holding.

The disp_hex module available on the course website was also super useful when you want to look at numbers or registers values. This was especially helpful when we were debugging the coordinate transform. But also the 8 leds were really helpful when it comes to enable signals.

Incident Blob Display(Emmanuel)

The incident blob module used arrays of variable depth (SHOT_LIMIT) to store the x positions, y positions and a parity bit to show that an array position has a bullet (to distinguish no bullet from bullet position 0,0). A 'generate' loop was then used to instantiate SHOT_LIMIT number of bullet blobs and the array was filled on trigger.



```
// use the array to make blobs
genvar i;
generate
  for (i=0; i<SHOT_LIMIT; i=i+1) begin: display_shot
    wire [23:0] current_pixel;
    bullet_blob draw_blobs(.x(bullet_x[i]),.y(bullet_y[i]), .has_bullet(has_bullet[i]),
                    .hcount(hcount), .vcount(vcount),.pixel(current_pixel));
    assign bullet_pixel[i+1] = current_pixel | bullet_pixel[i];
  end
endgenerate
assign pixel = bullet_pixel[SHOT_LIMIT];
```

The state of the module is updated at most once every half second (or any set delay time). The result is that when the user hits trigger and transmits the coordinates they aimed at, we display the new blob and every blob before it up till SHOT_LIMIT blobs. On 'clear', all the blobs are erased.

Text Display(Emmanuel)

The text display module was made by typing all alphanumerics [A-Za-z0-9] in Gimp and using matlab cropping tools to obtain the rectangle top-left coordinates, width and height of all the characters and stored them in a '.dat' file. We then wrote a matlab script to crop the letters using the provided coordinates with width and height of 31 pixels (enough to accommodate the widest and tallest characters). The retrieved pixels were then stacked into an array of (31*62 characters) by 31 matrix and then converted to a '.coe' file to be loaded into the BRAM.

To read a letter on vclock, we check if hcount and vcount were within the intended area and obtain the address in memory based on the alphabet number [0-61] corresponding to [A-Za-z0-9]. This part was mostly tedious because we manually got all the crop rectangles which probably could have been easier by looking up an already made bit representation of alphabets online. We also made two iterations of this module. First, we used bigger texts and stacked the images as 64 by (64*62). However, this orientation couldn't fit into the BRAM because the BRAM was much deeper than it was wide so we divided the matrix into four ranges of alphanumerics and used that for the display. After realizing the proper dimensions of the BRAM, we switched to (31*62 characters) by 31.

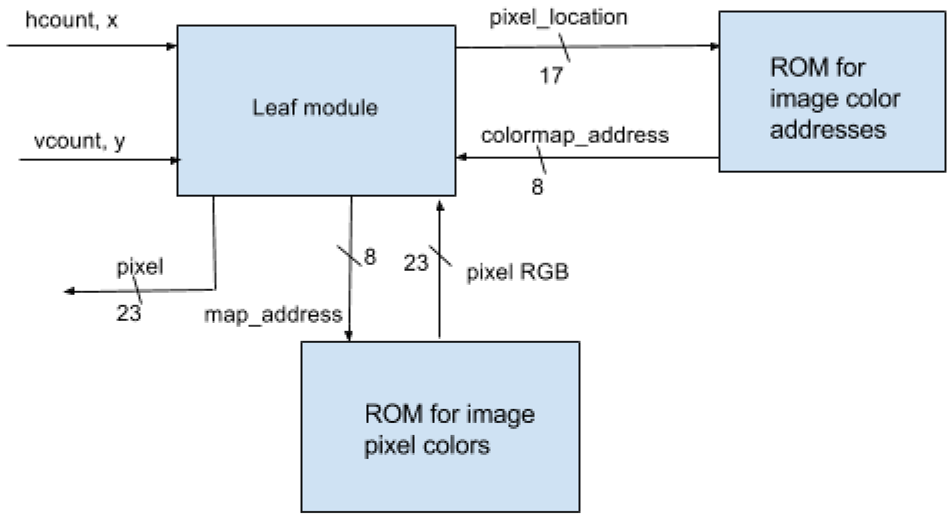
Score Display(Emmanuel)

The score display module took in the x,y coordinate of newly sent hit positions and determined the scores based on the relative distance from the center of the target. The score was then converted from binary to decimal with a maximum possible display of 999. The binary to decimal converter was as presented in Lpset 8. The ones, tens and hundreds digits were then sent to the text display module to display the score on the screen. The score was alpha blended to overlay the bottom of the target.

Keyboard Input Display(Emmanuel)

The keyboard module was as provided on the course website for Fall 2005. We supported alphanumerics, backspace, and enter (for resetting the typed characters). At the rising edge of keyboard ready signal, we stored allowed ascii values and sent the combination of texts to the text display module to display the letters. A pointer was used to determine where to insert a new character and on a backspace hit, we removed the most recent character. Since the register for the deleted character would have a zero value, we shifted all the character values up by 1 so that a zero value was an empty space (and an ‘A’ was represented by 1, etc). Of course, we accounted for this shift in the text display module. The keyboard input limited character display to 10 characters which was sufficient to display most user first names or nicknames.

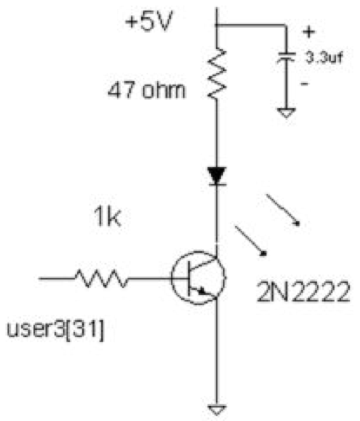
Immersive Background(Emmanuel)



To get the immersive background, we decided to use falling leaves to simulate an outdoor experience (or possibly a christmassy feel since we were close to that time of the year). We got an image of leaves with a black background to make thresholding easy for alpha-blending. Using the GIMP software, we cropped out part of the image to display. We then stored the result as a bitmap, addressed by 8 bits at each pixel location, making a total of 256 possible colors in the color map. We then loaded the image into the matlab script provided on the course website. The script loads the bitmap and stores the 256 RGB values in a ‘.coe’ file. We initially tried storing all the bits of the image as a 1D array in the BRAM. We expected the array of size about 1.5 MB to fit into the ~3MB memory. It however took very long for the array to load into the BRAM. We therefore use ‘imresize’ from matlab to resize the image to about 60% of the original image which was about 83 KB. We stored the image in a ROM in the BRAM and then for each pixel location, used the 8 bit address to get the color from the colormap ROM. To display multiple copies of the same image, we checked whether hcount and vcount were in the appropriate ranges and shifted the pixel address appropriately to start from zero at the beginning of each of the ranges. This strategy reduced multiple instantiations of modules accessing the same memory addresses, but the multiplications used to find the correct addresses for each range needed to be pipelined. A user of our game commented that the falling leaves lent our game a merry feel!

IR Transmitter(Emmanuel)

The transmitted data stream had a preamble followed by the x,y position on the screen, a trigger bit, and the last bit to tell whether clear had been pressed. The implementation was a 23 bit serial pulse width modulation (x,y,trigger,clear which made 11+10+1+1=23 bits). Most of the implementation was similar to that provided in Lab 5b. The major change required was to increase the wait time between successive transmissions and update the signals being sent over. The major challenge this module posed was that after using the transmitter for a while, it would get stuck in state “START” or “WAIT” and stop transmitting. We spent a lot of time with simulations and actual fpga runs trying to figure out the root cause. Ultimately, we couldn’t figure out the bug and resorted to a counter that reset the state to IDLE after it got stuck.



For the circuits, our Vishay TSKS5400S (Infrared Emitting Diode, 950 nm, GaAs) from Lab 5 worked only for ranges of up to about half a meter. To increase the intensity of the IR waves, we tried to increase the collector current to the NPN BJT so that the forward current through the IR transmitter would also increase. We erroneously tried for a while to reduce the 47 ohm resistor while keeping in mind that the maximum forward current for the diode was 100mA. There was a marginal increase in the range to about ¾ of a meter. The measurements from the oscilloscope showed an almost negligible changes in the voltage drop across the resistor connected to the collector of the BJT. After a number of variations in resistor values, we ended up changing the base current instead. We used the relation below:

$I_{c0} = \beta I_{b0}$
$I_{c1} = \beta I_{b1}$

This gave the relation:

$$I_{c1} = \frac{I_{b1} \times I_{c0}}{I_{b0}}$$

The new base resistance was used to regulate the new base current such that the new collector current was still under 100mA. With a collector current of about 90mA, we could reliably transmit up to about a meter and half. Eventually, my project partner brought in a stronger IR emitting diode which worked well within four meters range so we switched to that.

IR receiver(Emmanuel)

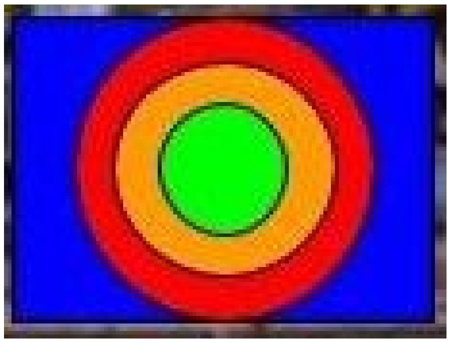
Infrared receiver (RPM7140-R) was used to detect and read the output of the IR transmitter. The signal received by the receiver was parsed according to the established protocol from the transmitter and the data was sent over to rest of the game.

The trigger signal was an interesting one. Each time we transmitted a signal with trigger, we added a trigger high to the packet. If we transmitted a clear, clear was high and trigger was low. One would therefore expect trigger to rise once during a continuous press of the gun's trigger. It however, rose multiple times. In fact holding on to trigger would continuously fill the array of bullet hit positions, which was designed to take a new hit information on the rising edge of trigger. We tried several attempts to get trigger to work properly. Some strategies included setting trigger to zero after trigger was released (after serial data stayed low for some time interval). We also tried using a buffer of about ten transmissions wide so that we stored all trigger values that passed since we started reading IR data within that window. We then used 'or' on all the bits to see if trigger had been high within the window to assume trigger had remained on. That also didn't work for a single long hold. We eventually used an FSM with a timer to undersample the transmission to say once every half second.

Sound(Mubarik)

The sound module was essentially a modified version of Lab5. We added a third and default mode where we play music from a phone by taking the data from the AC97 input and routing it directly back to the AC97 output. The two other modes were for record and playback from block ram, but the playback was signalled by trigger instead of playing back by default. When we pull the trigger the module read samples from block ram which was the sound of a gun fire.

Target Display Logic(Mubarik)



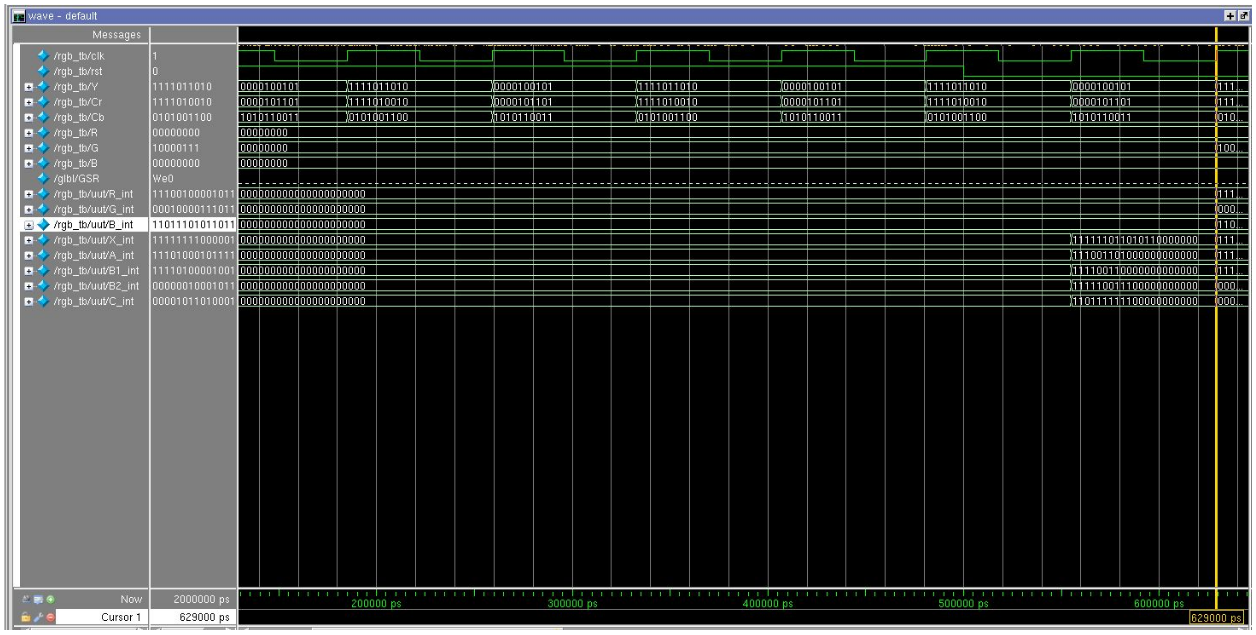
The target was a VGA monitor displaying a pattern of concentric circles. It was this pattern that we built our image detection on and used to determine score for a shot. Each circle had it's own score, and awarded higher scores as a shot got the closer to the center. We also displayed the bullet marks, score, the keyboard input, and blended image of falling leaves.

To make the game more challenging, we also added a mode which the circles move move back and forth horizontally. And everything to appear nicely with the pattern, we used alpha blending to mix the colors.

Game Logic(Mubarik)

We didn't end up making centralized game logic, instead each module had its own FSM to perform the required functions in sync with the rest of the logic. This was possible because the trigger and clear signals from the IR communication were signaling all the transitions.

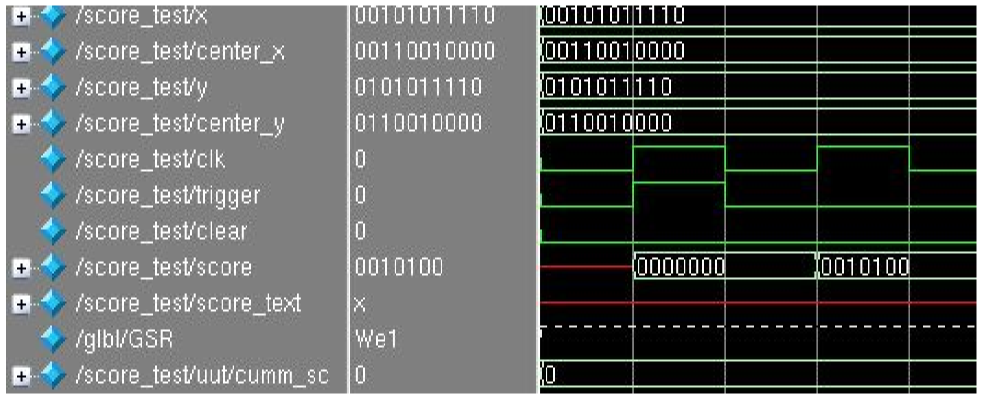
Testing



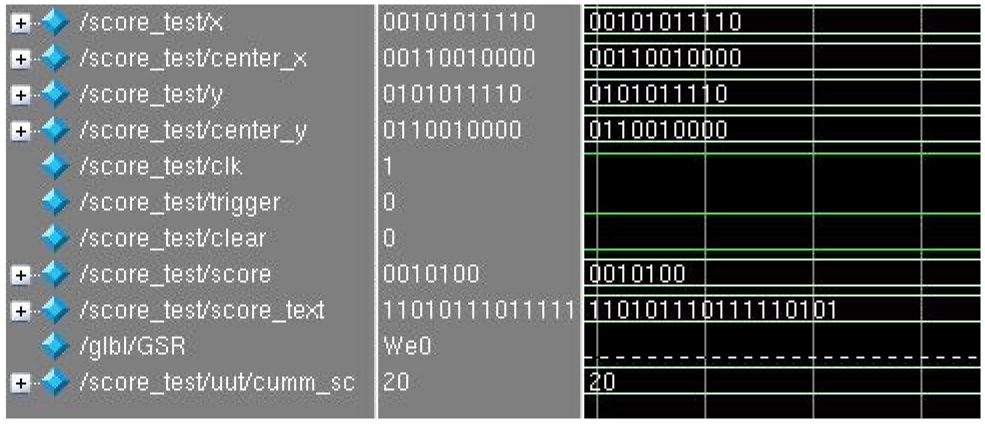
```
wave - default
Messages
/rgb_tb/clk          1
/rgb_tb/rst          0
/rgb_tb/Y            1111011010
/rgb_tb/Cr           1111010010
/rgb_tb/Cb           0101001100
/rgb_tb/R            00000000
/rgb_tb/G            10000111
/rgb_tb/B            00000000
/glbl/GSR            We0
/rgb_tb/uut/R_int    11100100001011
/rgb_tb/uut/G_int    00010000111011
/rgb_tb/uut/B_int    11011101011011
/rgb_tb/uut/X_int    11111111000001
/rgb_tb/uut/A_int    11101000101111
/rgb_tb/uut/B1_int   11110100001001
/rgb_tb/uut/B2_int   00000010001011
/rgb_tb/uut/C_int    00001011010001
Now                  2000000 ps
Cursor 1             629000 ps
```

Most of our testing happened on the FPGA, mostly displaying the signals of interest on the 7-segment leds, the eight other leds on the Labkit, and displaying images and video data on the VGA monitor especially earlier times when the code was taking about a minute to compile. However, we both used ModelSim for sanity check, and to see the delays of the modules we have borrowed from the course website before we upload the code on to the FPGA. The above waveforms of YCrCb to RGB is an example of such simulation.

When the code size grew, simulation became an increasingly useful and quick way for us to ensure we spent less time on fixing bugs later. The test below was for the score module. In the first image below, we found that the cumulative score stayed at zero even though the score was at 20. The score texts were also not updated. Through simulation, we realized that the score went to the bit to decimal converter at the rising edge of trigger, when it actually was ready during the next clock cycle. This was because, finding the score involved a multiplication that was ready for use a cycle later. This would have taken much longer to realize by displaying hex on the FPGA.



```
/score_test/x              00101011110
/score_test/center_x       00110010000
/score_test/y              0101011110
/score_test/center_y       0110010000
/score_test/clk            0
/score_test/trigger        0
/score_test/clear          0
/score_test/score          0010100
/score_test/score_text     x
/glbl/GSR                  We1
/score_test/uut/cumm_sc    0
```



```
/score_test/x              00101011110
/score_test/center_x       00110010000
/score_test/y              0101011110
/score_test/center_y       0110010000
/score_test/clk            1
/score_test/trigger        0
/score_test/clear          0
/score_test/score          0010100
/score_test/score_text     11010111011111
/glbl/GSR                  We0
/score_test/uut/cumm_sc    20
```

Challenges and Lessons Learned

We both ran into plenty of problems along the way. Even though we forgot most of them after we found solutions, we remember quite number of them. Many times, modules that were working as expected before just started failing. We came to learn that turning off the FPGA and reloading the files before you try anything else helped ensure our issues weren't hardware related. In the beginning problems like that would takes us long time to resolve but many of them became a second nature as the hours rolled on.

Mubarik:

Image processing was challenging and sometime frustrating but I learned a lot in this project. I continuously remained that the real world is not always consistent, and things don't work as expected. One clock cycle mismatch can make everything seem nonsensical. I would misspell one of the arguments to a module, my code would synthesize but the screen will turn black or would show some other weird patterns that made no sense. I often found myself asking, "what? What did I just do?".

Emmanuel:
While using BRAM for the first time, I would finish the setup and then get several errors about verilog not finding the modules for my ROM. I spent a couple of hours trying to find which of the generated files to include into the project to get it to work. After turning off ISE and coming back the next day, the code just compiled. This was an unexpected challenge that sunk some time. Of course afterwards, I did well to close ISE and reopened each time I added a new ROM. Again, as described in the IR transmission and receiver modules, the trigger would work fine for a while and unexpectedly stop working. Initially we thought it was an issue with the circuits (maybe failing components). After debugging for a while, we ended up finding some work arounds without really finding the cause of the erratic behavior. The text display module also showed some unexpected drift in the offset and displayed halves of two letters at the same time. Turning the FPGA off and back on fixed this issue.

Overall, I understood better the process of pipelining in verilog in addition to having a general idea of the concept of pipelining. Getting a chance to learn how to interface with devices including the keyboard, and re-learning some circuitry was fulfilling. My goals for taking this class included understanding tricky signals(glitching), synchronous and asynchronous designs and understanding how data was stored in and read back from memory, etc. This project helped me reach a solid understanding of these various concepts.

End Product and what we missed

Overall, we reached most of the goals,both baseline and stretch, we set forward and stayed interesting to us. At the End, our target detection was working with great accuracy and reliability, our wireless (IR) communication was almost flawless, we had beautiful display with blended background images, scoring was done and displayed properly, the boom of gunfire was playing in the background as we shot, and we allowed shooters to display their names using a keyboard. However, there are couple of things we wish we had time for before checkoff. The first of those is to create more immersive background then we had at end, but we also wanted to create more centralized game logic with more features than we had, including allowing multiple players to compete against each other.

To say the least, we are proud of what we have done and learned as a team. Our game is quite expandible and has the potential to be a more fascinating game to play and we would have loved to have gotten it to a much better state.

Possible extensions

According to our stretch goals, it would be interesting to make our game logic more complex by introducing moving targets. That would certainly introducing some interesting timing insights. Successfully integrating an IMU to work in conjunction with the image processing to give precise coordinates of the gun's heading relative to the target would improve the accuracy of hit positions. Especially so for the transformed coordinate frame in which the center of the camera determined the position of the hit blob. Since the said frame required more accurate aiming, having an IMU to work alongside the image processing to affirm our computations would improve our confidence in the system. For our immersive background, it would be interesting to have two or three sprites for about four leaves and cycled through them to simulate falling leaves that turned around during their fall. That would offer a more natural feel to our digital outdoor shooting range. We could also add a couple of added challenges to the game including randomly shifting the center of the target to make it harder for the user to hit the bull's eye. For aesthetic purposes, we could make the target 3d.

Acknowledgements

Special thanks to Prof. Gim, Joe Steinmeyer, Mitchell Gu, and the rest of the course staff for always being available and helpful. We received a lot of help along the way and we remember and are grateful for all of it. We were joking about how Prof. Gim has a lookup table in his head because he had a quick answer for every problem we ran into. Joe opened the lab and was available and helpful at times of rush and that really made the difference. Thanks to Mitchell Gu for guiding us along the way, always insightful comments to save us from pitfalls. A special acknowledgement to the generous contributors to the 6.111 store of useful modules who helped reduce the load of having to write all our modules from scratch.

Conclusion

Digital shooting range is a good pastime to help rifle and pistol lovers do what they love in the comfort of their rooms. We learnt a lot from this project because of the be exposure to image processing in hardware as well as very interesting scenarios of signal timing. The project also had room for stretch goals which allowed us to also learn more about signal processing, image display, keyboard integration, to mention but a few. Our hope is that our users give us sufficient critical feedback on how we can improve the product and that they will have as much fun as possible in a safe digital shooting range.

References: GitHub