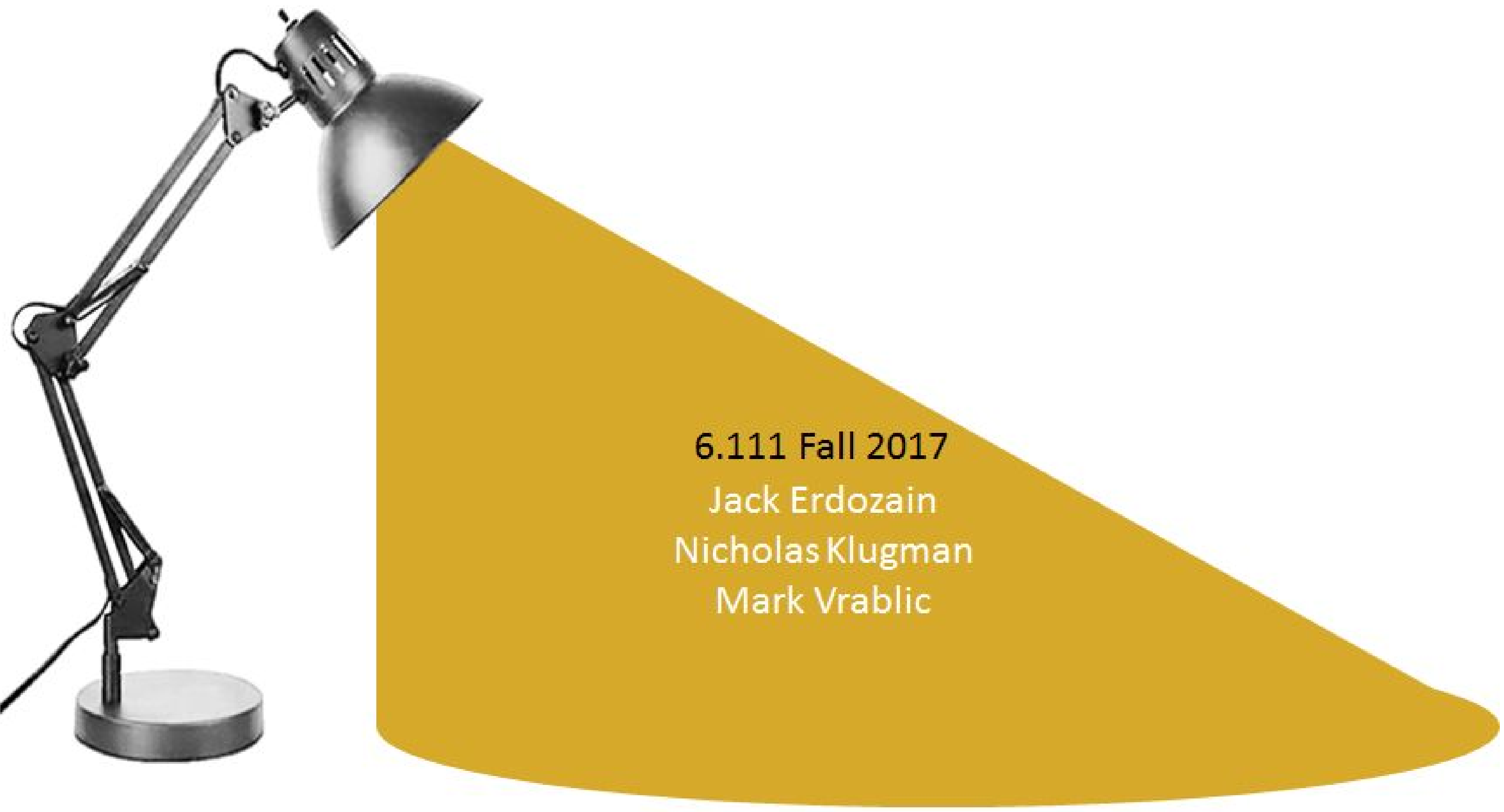
FPGA Robot Arm Assistant

Final Project Report



6.111 Fall 2017

Jack Erdozain

Nicholas Klugman

Mark Vrablic

Table of Contents

1 Introduction

1.1 Project Inspiration

This project was inspired by LuminAR, a project by the Fluid Interfaces group of the MIT Media Lab. The project was a striking new take on human interfaces through robotics. The project later spun off into a fairly successful company, named Tulip, still operating in the Boston area providing computer vision based assistive interfaces for manufacturing.

From the LuminAR website[1]:

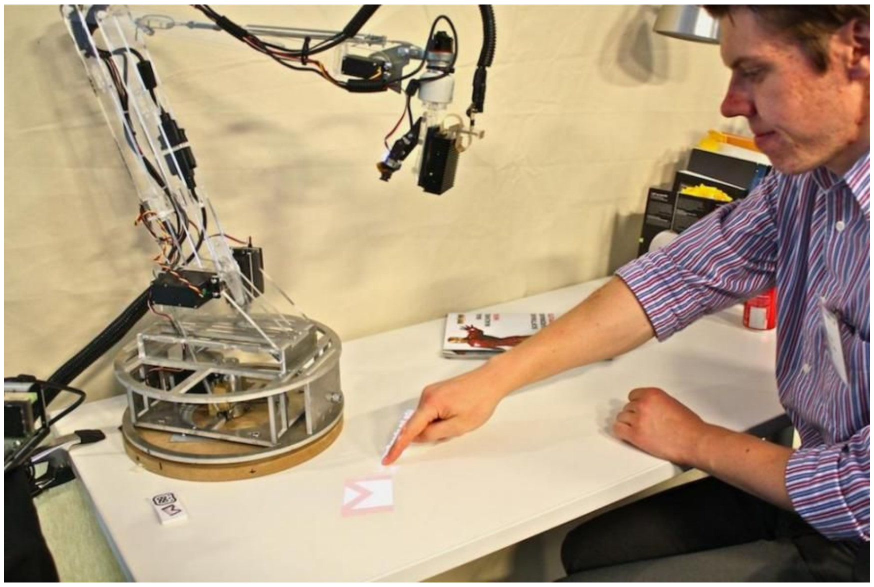


Figure 1: LuminAR Demonstration

LuminAR reinvents the traditional incandescent bulb and desk lamp, evolving them into a new category of robotic, digital information devices. The LuminAR Bulb combines a Pico-projector, camera, and wireless computer in a compact form factor. This self-contained system enables users with just-in-time projected information and a gestural user interface, and it can be screwed into standard light fixtures everywhere. The LuminAR Lamp is an articulated robotic arm, designed to interface with the LuminAR Bulb.

1.2 Project Overview

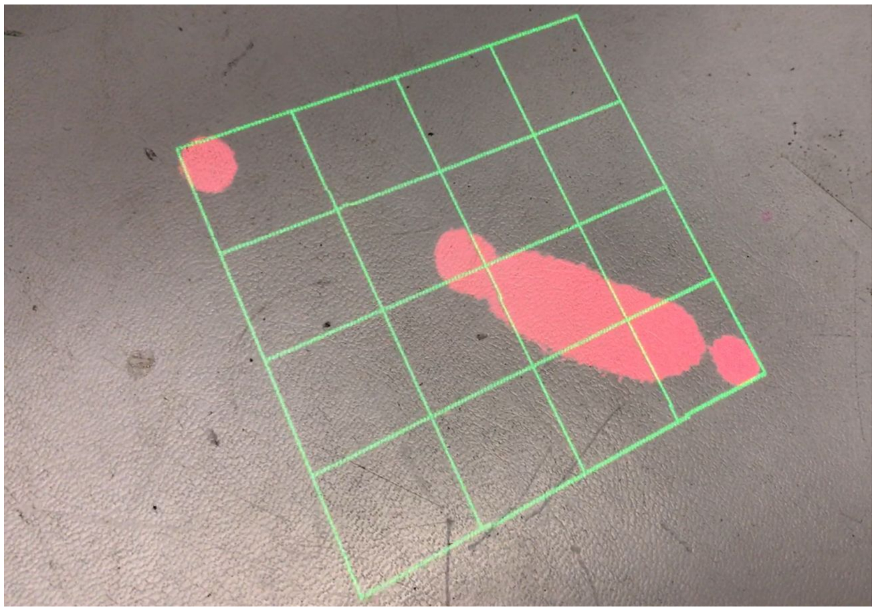
The goal of this project was to project image and text notifications on a desk and interact with that projection via computer vision. The physical hardware platform for the

[1] http://fluid.media.mit.edu/projects/luminar

project was a robotic arm, disguised as a common desk lamp, with a camera and projector at the end, all controlled by an FPGA. Notifications come from from a connected computer and are keystone-corrected when projected, to eliminate distortion caused by projecting at an angle. The object that the computer vision tracks for the projection to follow is an array of IR LED's, henceforth known as the "puck." Rotating the puck scrolls through notification text and clicking the center of the puck will switch between modes.

1.3 Goals

1.3.1 Computer Vision Goals



The objective of the computer vision modules are to produce useful tracking information for the robotics modules to utilize in following the puck and to determine user interface information from the puck (in this case the angle of the puck and if the user is creating a click) to be used by projection modules. Identifying a puck with a low resolution camera against an unpredictable environment is difficult and goes into higher levels of computer vision techniques outside the scope of this project. Instead we engineered the situation to make a more robust solution. By removing all visual light from the camera, and placing IR led's on the puck we were able to derive the center of the puck in respect to the field of view of the camera, creating an "error" value for our robotics feedback loop. Similarly we were able to bound the contour of the IR identifiers and use this information to systematically check for the angle of the puck and if it was being clicked. Varying levels of angle detection resolution were implemented.

Additionally a VGA output of the camera's vantage point alongside a "heads up display" like overlay was created to assist in debugging and as a visual demonstration of what was happening internally.

1.3.2 Robotics Goals

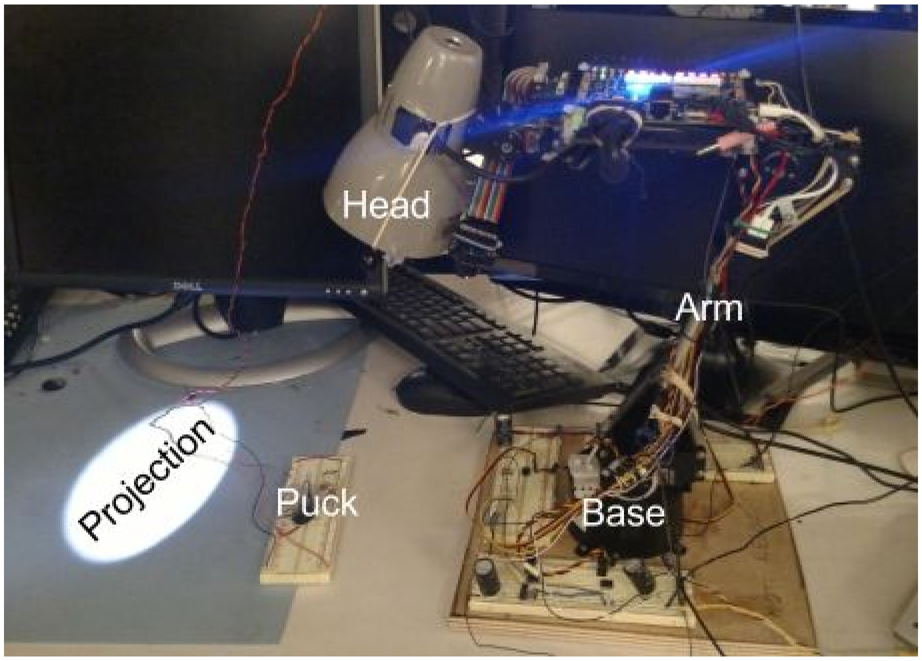


Figure 2: Labelled Assistant Setup

In order to track the puck, the robotic arm must move to minimize the error between the detected puck position and a desired puck position that remains fixed in the camera's field of view. The desired puck position is representative of a desire to maintain the camera (mounted on the head of the arm) in such a position that the puck is centered in the camera's field of view. This goal is achieved by commanding a change in the angle commanded by the arm's servos proportional to the detected error. To throw the projection farther from the arm or bring it closer has three degrees of freedom in our arm, since there are two servos in the arm and one on the head that can all be used for this purpose; another goal of the robotics portion of this project was to utilize these degrees of freedom to prevent the arm from over-extending itself to a position from which it cannot come back (which can happen because of the weight of the projector and FPGA close to the head of the arm). Beyond just tracking the puck in the two dimensions of the surface, we also endeavored to minimize unnecessary movement or "jiggle" by placing minimum error bounds before movement.

1.3.3 Projection Goals

The biggest challenge of using a projector on a robot arm in motion is maintaining a rectangular image on the projection surface. If the image were projected normally, the angle would cause the further away parts of the projection to appear larger, and the closer ones smaller. Luckily, however, this warping can be corrected by manipulating the image before projection. This transformation allows the image to look correct at the cost of resolution being lost. What was a straight line of pixels may become diagonal which cannot be projected perfectly (figure 3), and there will always need to be black borders on at least one side of the screen to maintain the correct aspect ratio. Additionally, on regular projectors, the edges of the image will be out of focus. Laser projectors do not have this issue and are compact, so it made sense to use one for this project

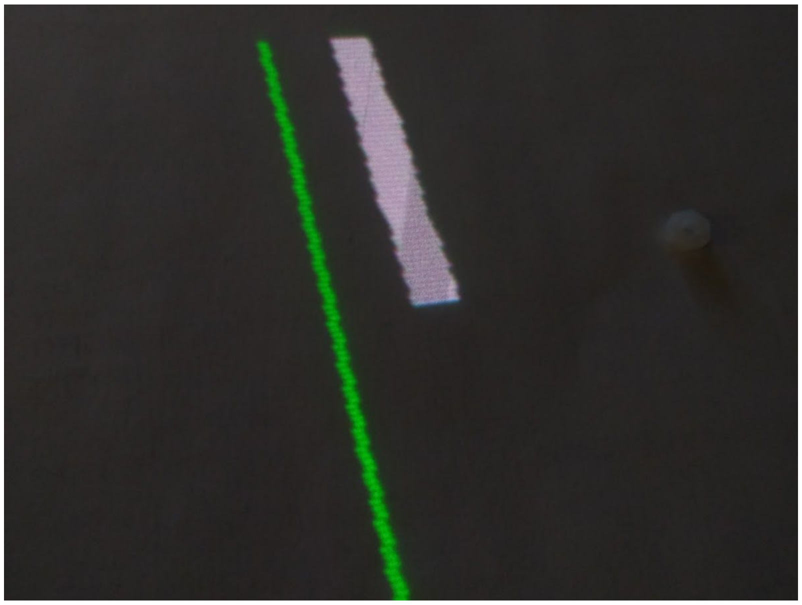


Figure 3: A vertical line close-up after keystoning, note the lost sharpness

This image transformation to maintain a rectangular image is called keystone correction, and the type of transformation used is known as a homography transformation. It is a mathematically intensive transformation, however, requiring a system of 8 equations obtained from a matrix multiplication described further here[2]. This large system is unnecessarily complex to solve on the FPGA itself, even after simplifying it based on our physical constraints, as it requires lots of precise division. Therefore, this project uses a different approach better suited to our constraints and the

[2] http://www.corrmap.com/features/homography_transformation.php

strengths of an FPGA. This approach is explained in detail with the keystone corrector module in the Implementation section.

In order to make the keystone correction worthwhile, cool things are displayed on the projector. Four regular VGA programs feed into the keystone corrector:

1. Notification program
    - This program displays incoming messages from a computer connected with serial running a python script (Appendix 6)
2. Spotlight program
    - The projector is housed in the shell of a lamp, and this allows it to retain functionality as a lamp
3. Image of the Building 10 dome
    - This is to show off the accuracy of the keystone correction and the python-based image to verilog 12 bit sprite converter
4. Pong
    - What self respecting project doesn't play pong? Additionally, since it was from lab 3, it could be used for testing during initial development of keystone correction

The robot arm looks somewhat alive when it moves, so to give it a personality, adding sound effects was made a stretch goal.

2. Design

2.1 Physical Components

- **Robotic Arm:**
  An off the shelf robotic arm capable of holding a small pico projector. It utilize 4 Servo motors, one to rotate the base, two to angle the “body” of the and one to rotate the head.

- **Pico Projector:**
  A laser projector was used as it was light weight, small, does not need to be focussed based on distance, and can was available thanks to Joe.

- **Puck:**
  An object (in this case a breadboard) with IR LEDs which can be moved while still easily being viewed by the camera. This puck used 7 LEDS, each powered at 5 Volts, each with a current limiting resistor at 300 ohms.

- **Camera:**
  An NST camera with a thin visual lighter filtering film(pulled from a floppy disk) layed over the lense.

2.2 Design Challenges

- **CV**:
  Because the robotics control and feedback loop relies on the CV, the solution must be robust and low noise.

- **Keystoning:**
  The mathematical operations for keystone correction are quite hard so an alternative to this raw computation must be found.

3. Implementation

3.1 Overall Block Diagram

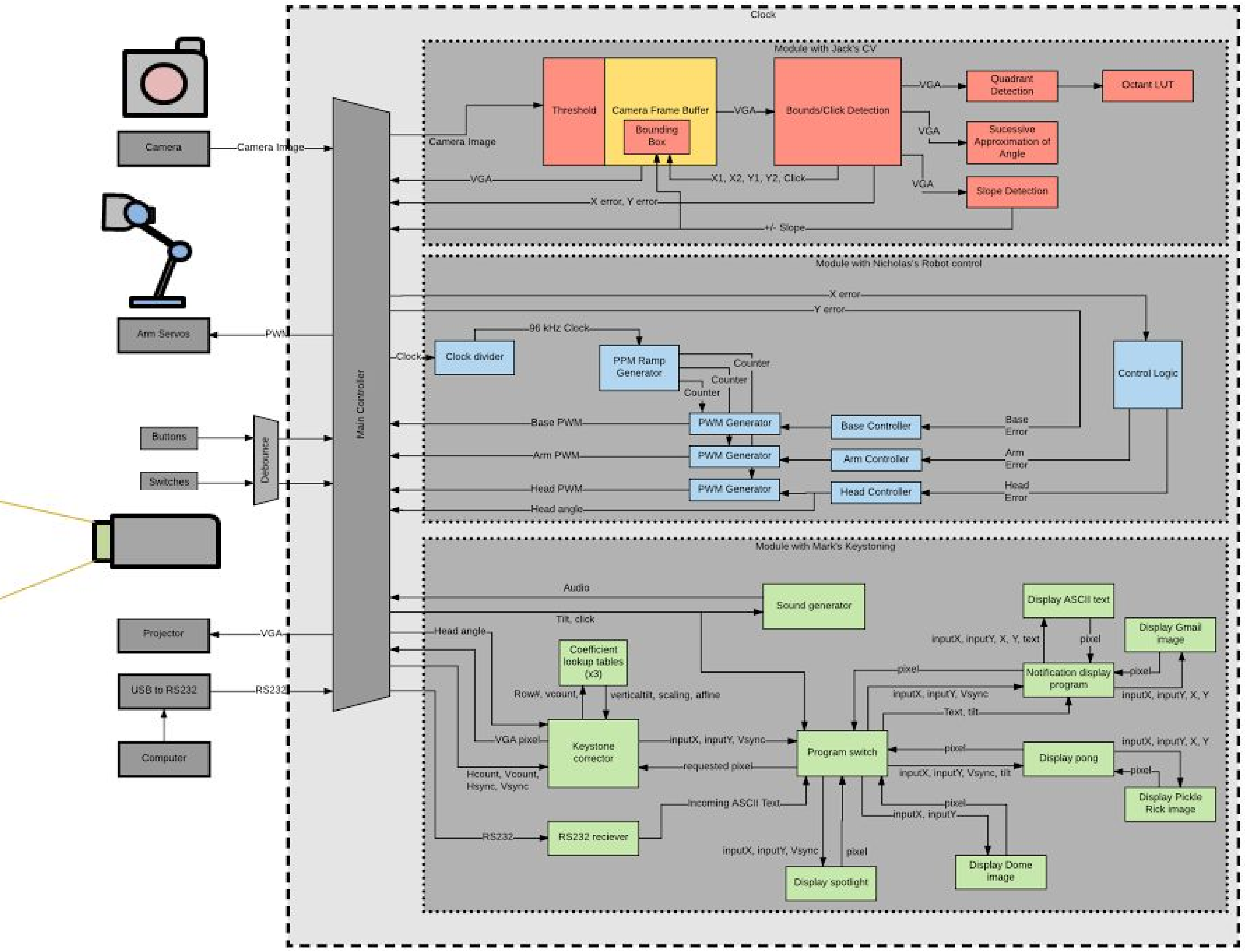


Figure 4: Overall Block Diagram

3.2 Modules

3.2.1 CV Modules (red)-Jack Erdozain

Camera Frame Buffer

As a foundation for the Computer Vision modules the example camera frame buffer created by Weston Braun was used and modified to meet the needs of this project. At a high level this code took in a camera video feed and produced a VGA output. As a first step all of the information from the frame buffer was masked to only appear as red (for debugging purposes). Similarly the frame buffer was passed through a binary threshold filter. If the intensity of the pixel in question did not surpass a certain intensity the pixel was ignored, and turned black, aka a “0”. If it did surpass the threshold it was considered a “1”. In addition a bounding box identifying the edges of the IR image and was layered on top of the VGA feed in green or blue. This was used for diagnostics purposes and as a demo of what the robot “sees.” More detail will be given on this bounding box in later sections.

Bounds/Click Detection

For the “error” to be generated for use by the robotics modules the center of the object being tracked must be correctly identified. A simple way to approximate the center of an object is to identify its farthest most edges and then mark the location of half width of the edges plus the location of the leftmost edge, and half the height of the edges plus the location of the top edge. The following is an example of what the CV would output on its VGA when detecting the center of a circular IR shape in front of the camera.

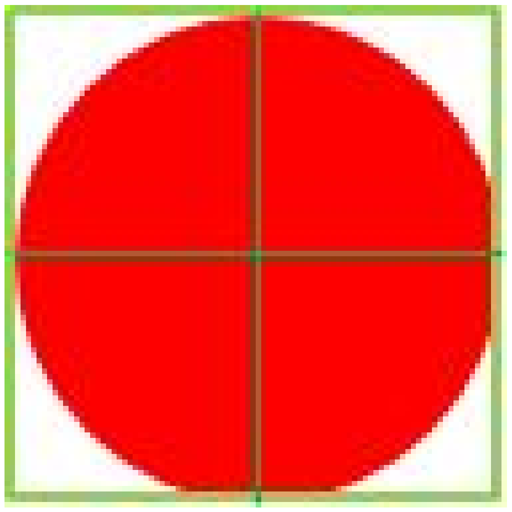


Figure 5: Simple object bounding

To find this bounding box the output of the VGA was monitored. To find the y edges was simple as the progression of y values is monotonic with time on a VGA line. The top edge was identified by marking the first pixel that was above the threshold. The y location was stored from this as the minimum y. To get the bottom edge the last y location that passed the threshold was identified and stored as the VGA output was rastered. The x edges were not as simple though as their output is not monotonic (rows are rastered consecutively as the image is drawn). To identify the x edges the code would note every time a valid pixel was found. If that pixel corresponded to an x value that was lower than the lowest one currently identified it would be stored as the new lowest identified x. The same logic was applied for finding the highest possible x lpixel. When the VGA rastering was complete (hcount = 481 and vcount = 641) these identified values were transmitted to other modules alongside the value of the center of the box which involved simple algebraic operations on the edges that were detected. Additionally an error value was calculated for the robotics feedback based upon this center point and a hard coded ideal center for the puck.

One thing to note about this procedure is that to identify the X values it is important that the starting conditions are carefully selected. The starting value of the register for the leftmost x edge (in the code this register is named lowest_x) is initialized to 641, an impossibly high value. In this case if any pixel is identified as valid it will assuredly have a lower x value and the register will be reassigned. The same logic is applied to the rightmost x edge register (in the code it is named highest_x). It starts with a value of 0 so that all the pixels identified will at least replace the initialization value. After a VGA rastering is completed these registers are reset to these values for the next rastering and detection cycle.

One interesting result of this is that it is easy to identify an invalid state or if no pixel was detected on the camera. If the left edge has a greater value than the right edge or the top edge has a greater value than the bottom, it is considered an invalid state and the robotics module is given an error of 0 to bring the robot to a stable position. In this way if the puck is removed suddenly, the arm will hold its position.

Now that we were able to detect the bounds of an object and its center it was important to implement the clicking interface for the projection modules. The puck was expanded from a single IR LED, making a circle, to a string of 3 LEDs, making 3 equally spaced circles along a line. An example puck placed at 4 different angles with a bounding box and center cross hairs is shown below.

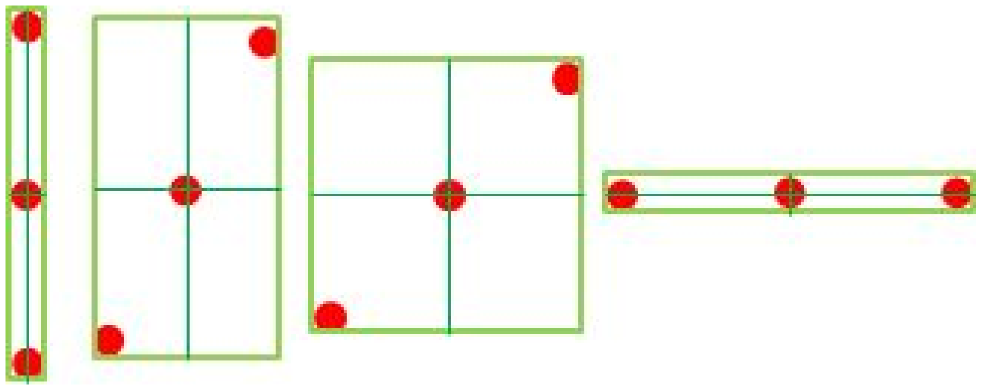


Figure 5: Center detection independent of angles

It’s worth noting that no matter the angle of the puck the bounding box will always produce a center that overlaps the center IR LED (the width of the LED’s projection gives some tolerance for error). To implement the clicking from a user the code would detect the presence of valid pixels in a 3x3 box at the center of the crosshairs. If even a single pixel was found inside that box the LED was detected, otherwise it was identified as a click. The advantage to this method is that the bounding box does not change as a result of the center LED being covered and therefore does not interfere with the robotics tracking.

Slope Detection

It is worth noting that the puck geometry described above will not change its center location independent of rotation and clicking. What will change based on rotation is the ratio of length to width. In this ratio the angle is encoded. More detail on this included in the section “Successive Approximation of Angles.” As a simpler and more robust first step a method was implemented for detecting if the slope was positive or negative. The code was also set to change the color of the bounding box displayed on the VGA line based on if the slope was considered positive, green, or negative, blue. An example is shown below:

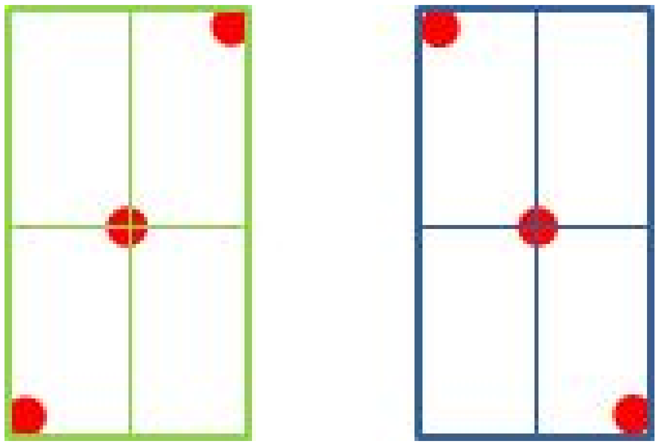


Figure 6: Puck slope, positive and negative

One way to detect this slope direction is to identify if the top circle lies to the right or left of the center circle. To do this another register was added. This register, during the VGA rastering process, would identify the Y location associated with the pixel that marks the leftmost X edge. If the value of this register is greater than the value of the Y location of the center circle then it must be a positive slope (as shown on the left), but if it's value is less than the center circle it must be a positive slope (as shown on the right). This turned out to be an extremely robust way to identify this property. This signal was used by the projection module to be used for features such as moving the pong paddle and scrolling through text.

Quadrant Detection

Though detection of positive or negative slope is effective in creating a binary input, which was the extent that our demo required, it was determined feasible to make a system that was capable of correlating angle based upon the slope ratio of width to height of the bounding box. The problem is that the bounding box can only identify the angle in respect horizontal axis. For example if the puck is continuously rotated from 0 degrees to 360, identifying the angle would only produce an output 0 to 90 with repetition (the output is modulo 90). This is because 3 LEDs in this pattern cannot produce a uniquely identifiable object in respect to angle around the entire unit circle. To solve this problem the puck was changed in such a way that the aforementioned modules would still function in their entirety, but such that a unique pose could be ascertained. The resulting puck became the following:

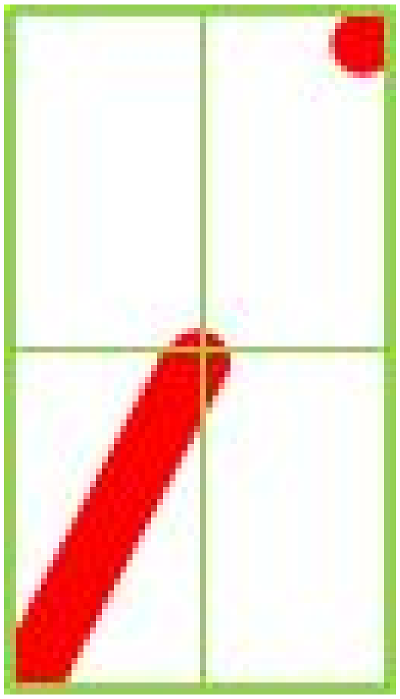


Figure 7: A puck identifier, valid for all angles

Its should be apparent that this shape can be identified at separate angles as it traverses from 0 to 360. The ratio of width to height of the bounding box will still encode the angle, but now we can identify which quadrant this angle is in by finding where the solid strip pixels lye. To do this pixel detection was set up at the center of the four quadrants of the bounding box. If IR light lit up this pixel it was safe to assume that the stip had landed in this quadrant. For example the image above produce an output of q1 = 0, q2 =0, q3 = 1, and q4 = 0 as it corresponds to the the strip landing in the third quadrant. With this information and the ability to generate a 0-90 degree angle from the bounding box it is possible to create an angle from 0 to 360 degrees.

Shown below is an example of the CV module recognizing an IR puck (on the right) of this form factor, projecting down what it sees and placing crosshairs on the 4 quadrant centerpoints it uses to check for the strip.

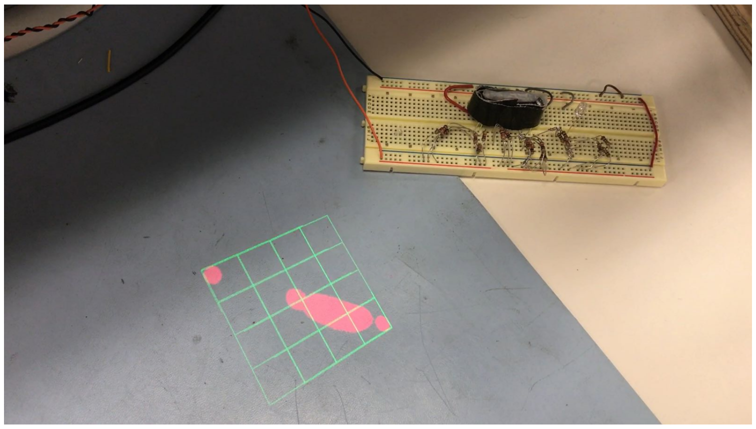


Figure 8: Quadrant detection output overlayed next to the puck

Octant LUT

Though the above method is effective in localizing an angle to the quadrant it belongs, practical constraints limit its effectiveness. The IR LEDs have a physical width such that when the CV module detects them and applies a bounding box to the shape, it is impossible for the box to ever be smaller in width or height than the diameter of one of the LED's projection onto the camera. This means that at 0, 90, 180, and 270 degrees a slight angle above or below that number will be detected, as a true horizontal or vertical require a width or height of 0. To ensure that these specific, but very important angles can be detected the octant LUT was created. This LUT takes in the values of the binary pixel detection of quadrants 1, 2, 3, and 4 from the quadrant detection module. The premise is that if the puck is horizontal or vertical two quadrants will detect a pixel simultaneously and all other angles in between will only have one quadrant detection at a time. This is shown below by the puck being rotated in steps of 45 degrees.

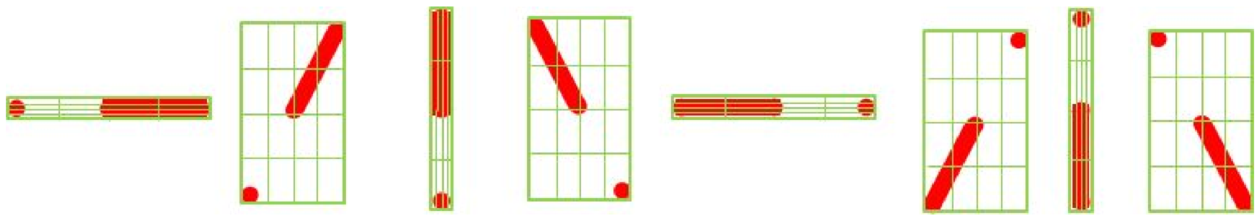


Figure 9: Octant detection from 0 to 360 degrees

This produces a LUT of the following:

q1	q2	q3	q4	output
1	0	0	1	0 dgrees
1	0	0	0	quadrant 1
1	1	0	0	90 degrees
0	1	0	0	quadrant 2
0	1	1	0	180 degrees
0	0	0	1	quadrant 3
0	0	1	1	270 degrees
0	0	0	1	quadrant 4

All other values not listed are considered invalid states.

Now that we have identified the quadrants and horizontal/vertical poses the remaining task is to approximate the angles based on the ratio of width to height.

Successive Approximation of Angles -Nicholas Klugman

For the scope of this project, a precise angle of the puck is unnecessary, however we did think of (yet did not fully implement in Verilog) an algorithm for determining the slope in approximate magnitude as well as sign. The algorithm is rooted in bit shifts and comparisons to narrow the slope down to ranges between successive powers of 2. By counting the difference in number of leading zeros between the x and y values from which the slope is being determined, the an upper or lower bound on the slope can be determined. Which type of bound it is, depends upon the comparison between the larger value and the smaller value then bit-shifted left by the difference in leading zeros (that is, shifted such that the first 1 is in the same place for each digit, or comparing the numbers after the first 1). This also give the opposite bound as one power of 2 higher or lower, depending upon the result of the comparison.

For example: x = 6’b110110=54 and y = 6’b001001=9. We know from division that the true slope is 54 / 9 = 6, which corresponds to 9.5 degrees off the x-axis. There are 2 more leading zeros in y than x, so our bound will be $2^2 = 4$. y << 2 = 6’b100100 < 6’b110110 = x, so this is a lower bound, which means that the upper bound is $2^{(2+1)} = 8$. Simply by counting zeros, bit shifting, and comparing, we have narrowed the slope down to the interval of 4 to 8, which corresponds to 7 to 14 degrees. In fact, for a screen size of 640 x 480, this first pass approximation gives us 21 different intervals in each quadrant and a maximum interval size of 18.4 degrees (see Appendix 1 for MATLAB script that demonstrates the algorithm mathematically and produces these metrics and graphs).

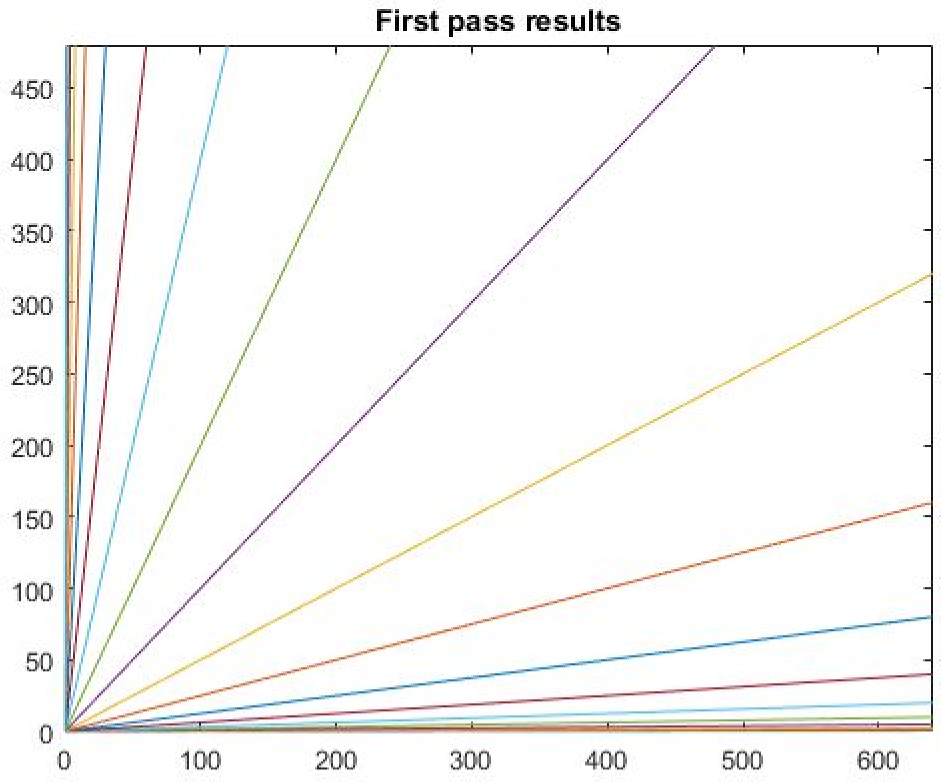


Figure 10: First pass slope approximation

By eliminating the shared leading 1 in the originally larger and newly shifted x and y values, the same algorithm can be run again to provide greater resolution within each interval. The regions with highest resolution in the first pass resulted from the largest shifts, which means that there is less information that is left to be gained from them, so the second pass evens out the resolution across angles. With the same screen size, it produces 89 intervals in each quadrant, with a maximum interval size of

2.2 degrees. Conversion from slope to angle requires a look-up-table with only 89 entries, one for each interval, not a full arctangent implementation. With only two passes of counting digits and bit-shifting, we would able to produce slope information with resolution of almost 2 degrees without implementing a divider.

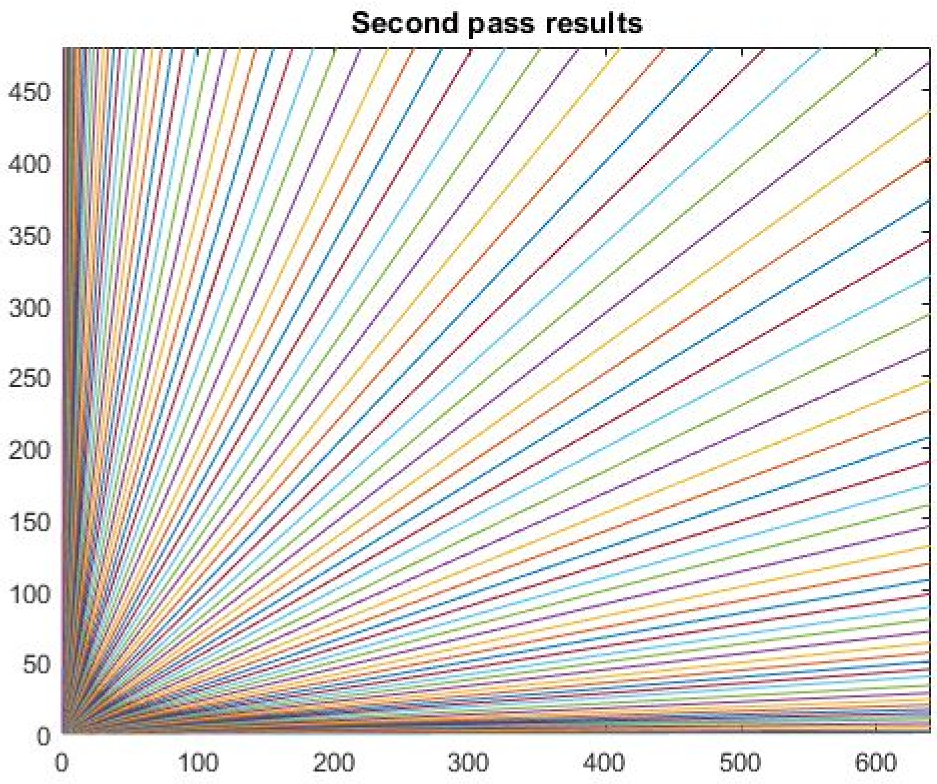


Figure 11: Second pass slope approximation

Because the slope is being converted to an angle the fact that as the slope gets steeper the distance between powers of 2 gets much larger (256 < slope < 512 seems less helpful than 1 < slope < 2), is mitigated by the fact that once the angle gets close enough to 90 degrees or 0 degrees, small changes in the angle lead to large changes in slope. In fact, this algorithm provides higher resolution in angle close to the axes than close to 45 degrees, even though the resolution in slope is highest close to 45 degrees (see Figure 10).

3.2.2 Robotics Modules (blue) - Nicholas Klugman

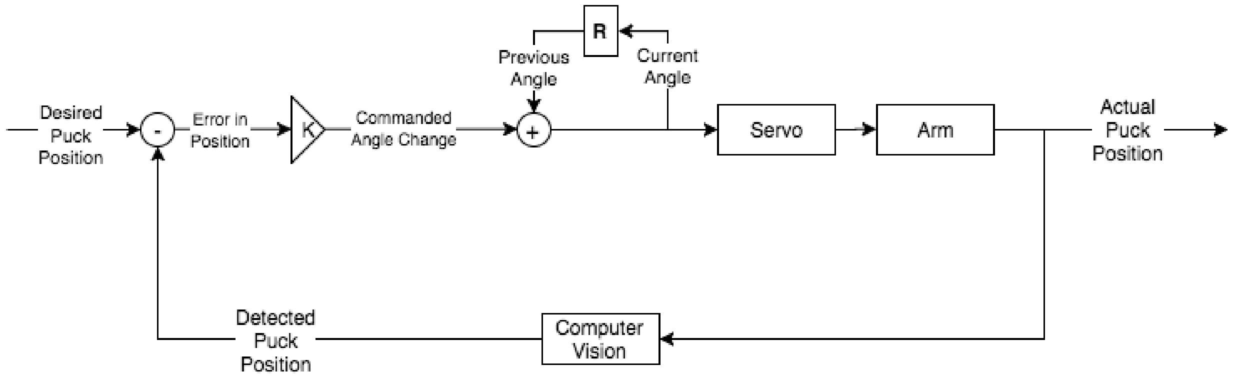


Figure 12: Robotics Feedback Loop Diagram

The robotics portion of the project is primarily concerned with the above feedback loop. In order to follow the puck, the arm works to maintain the puck in the center of the camera's field of view, which is the desired puck position in the diagram. The error in position is then calculated and scaled. If the servo command were angular velocity, then this scaled error could be fed in for a true proportional controller. In reality, however, the servo control is an angle command, thus the nested feedback loop to convert error to angle. Without that tiny inner feedback loop, an error of zero would command an angle of zero, when it should really be commanding a change in angle of zero.

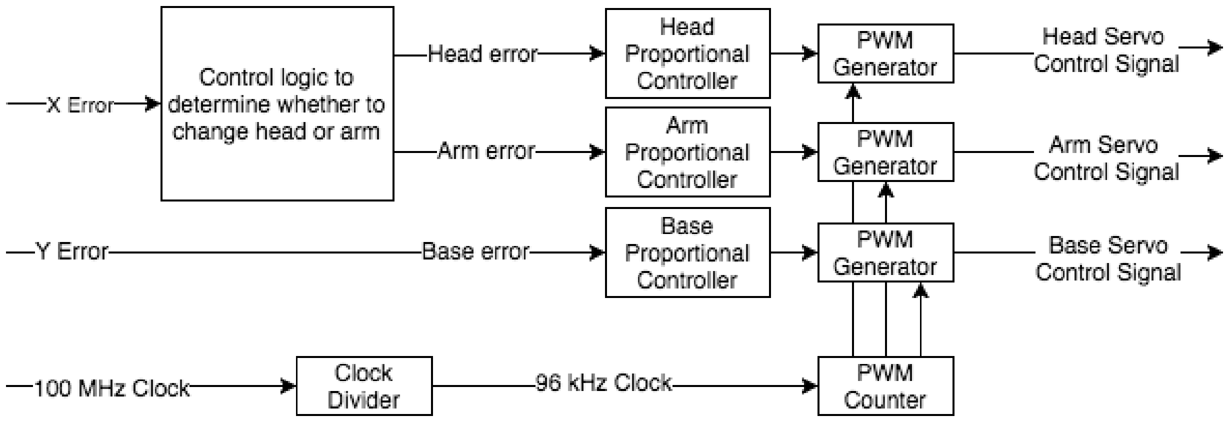

X Error
Control logic to determine whether to change head or arm
Head error
Arm error
Y Error
Base error
Head Proportional Controller
Arm Proportional Controller
Base Proportional Controller
PWM Generator
PWM Generator
PWM Generator
Head Servo Control Signal
Arm Servo Control Signal
Base Servo Control Signal
100 MHz Clock
Clock Divider
96 kHz Clock
PWM Counter

Figure 13: Robotics Block Diagram

It is worth noting that because the camera is mounted on the head of the arm, and therefore moving with the arm, any error in the x direction detected by the camera, always indicates motion radially away or towards the base, and any error in the y direction of the camera indicates motion around the base. For this reason, y-errors can be corrected by altering the base angle alone and, likewise, x-errors can be corrected by altering any combination of (including only one of) the arm angles and the head angle. The robotics block diagram only indicates one arm servo control signal because (as will be discussed in the next paragraph) the two arm servos are commanded to the same angle.

Not captured in the feedback diagram is the decision to move the head or the arm when an error in the camera's x direction is detected. The original scheme for that decision-making was to always adjust the arm and leave the head angle unchanged. The two arm servos would be set to the same value, thus keeping the upper portion of the arm (on which the FPGA is mounted) even and parallel to the ground. This scheme, however limits the range of the projection. It also would require the arm to extend farther than is safe. After extending far enough, the mass is so distributed as to prevent the base from rotating as well as to draw exorbitant amounts of current (enough to current-limit the power supply, thus us powering the four servos off of two power supplies). In order to keep the center of mass farther back, the arm is not allowed to extend past 90 degrees (still with the two arm angles equal). If the arm is in this upright position and the error is such that the projection should still be moved away from the base, the head angle is altered for a shallower angle, thus throwing the projection farther. When the puck returns to a position closer to the base, the head steepens its angle, until it is pointing almost straight down, at which point the arm begins lean back once more, so that the assistant is never projecting down upon itself.

In Figure X, each error is fed into a proportional controller block, which is the gain and the small feedback loop from Figure (X-1). This block stores an angle, which can be reset to a predetermined value by the press of a button. The angle is updated by adding the scaled error every clock cycle, leading to a change in the angle command proportional to the error. The constant of proportionality was chosen to be a power of 2, such that the scaling could be implemented by bit-shifting. In fact, the angle is stored with much greater precision than can be commanded and the error is added to these very insignificant bits, with only the eight most significant bits being sent out of the module as the angle. Since the clock is fast compared to the changing error, the delay seems much longer than a single clock cycle and the constant of proportionality seems larger than the number of bits shifted would let on, without need for a multiplier or a clock divider, also with the added advantage of potentially smoother control of the servos.

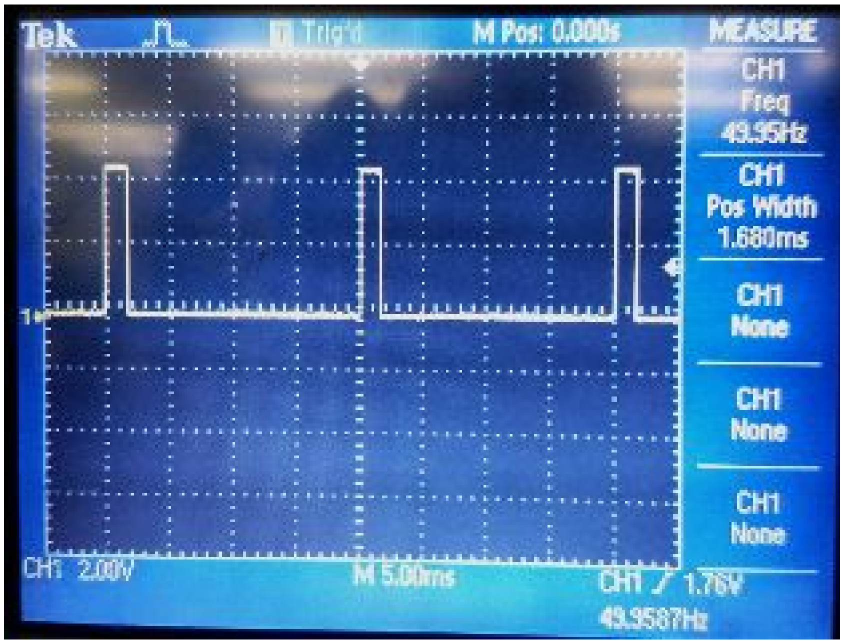


Figure 14: PWM servo command signal

The servos expect commands in the form of a 50 Hz pulse-width modulated (PWM) signal, which is high for at least 1/32 of the period and at most 1/8. In order to provide all 180 degrees of resolution it would have to be that 180 clock cycles = (1/8 - 1/32) * 1 / (50 Hz). This evaluates to 96,000 clock cycles = 1 s or a clock period of 96 kHz. So, the 100 MHz system clock was divided down to a 96 kHz clock for the servo controls. A counter that refreshed every 1920 clock cycles or at a frequency of 50 Hz (96 kHz = 1920 * 50 Hz), was then generated. To generate the PWM output from a commanded angle, the counter is compared against 60 + angle (60 = 1920 / 32), and remains high until the counter is greater than that threshold. There is also a check that the counter is less than 240 (240 = 1920 / 8), otherwise the output is forced low, regardless of the angle command. This additional check ensure that the PWM control line is not on for longer than 1/8 of the period.

3.2.3 Projection Modules (green) - Mark Vrablic

Keystone Corrector:

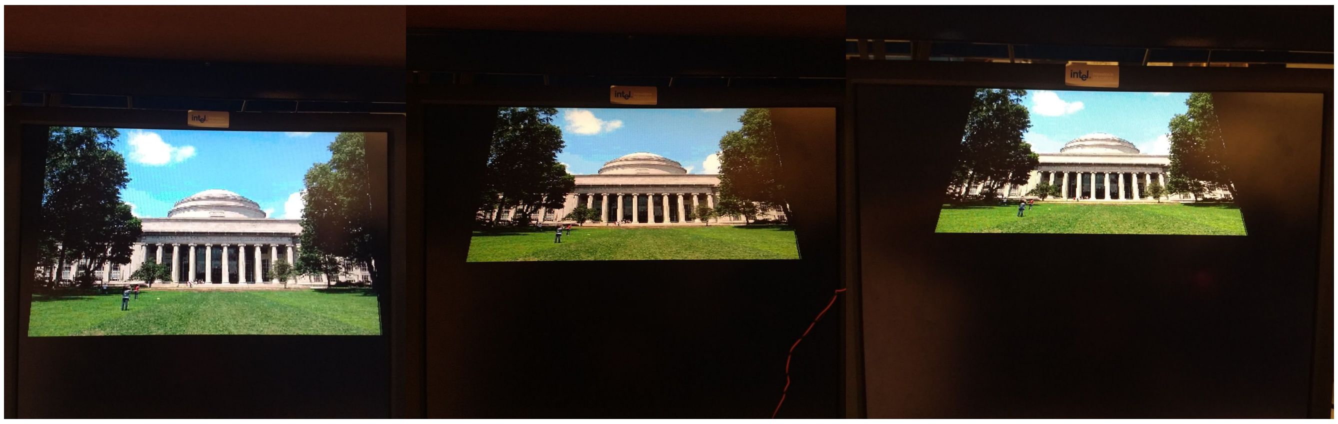
This module performs the keystone correction on a received VGA signal. Keystone correction is a complex combination of transformations which can be represented by large matrices. These calculations are very resource intensive for an FPGA to perform, however, as they require lots of precise division and multiplication. Luckily, the projector in this project is fixed on its horizontal axis, only rotating to point up or down, creating a trapezoid-like homographic transformation. This is much more simple to calculate, as it doesn't require matrices. It has been implemented in this project through a combination of three simple transformations:

1. Altering the angle which the trapezoid follows
2. Compressing each row by a number of pixels based on the angle and vcount to accommodate the trapezoid's width
3. Compressing the top of the image and stretching the bottom of the image based on the magnitude of the angle

The mapping required for which pixels to keep in the first two transformations was found with a linear mapping of rows with n pixels skipped using a python script. The amount of compression to apply was found using an unkeystoned pattern and physical measurement at 0, 15, 30, and 45 degrees. The measurements were then plugged into Matlab to form a quadratic equation for the approximate magnitude of compression to apply at any arbitrary angle.

Using this data to apply the transformations, generated hcount and vcount values calculated from the real ones are fed into unkeystoned VGA modules attached to the keystone corrector. The result is a new, keystone corrected VGA signal of another module based on the angle of the projector.

Figure 15: The dome with different amounts of keystone



Coefficient lookup table modules (x3):

The keystone correction works by looking up which pixels to skip, and these 3 modules are just big case statements for referencing which pixels to skip in horizontal rows, how much to slant the image at a given angle, and which vertical rows to skip or repeat as explained earlier. (see Appendix 3, 4, and 5 for code to generate these)

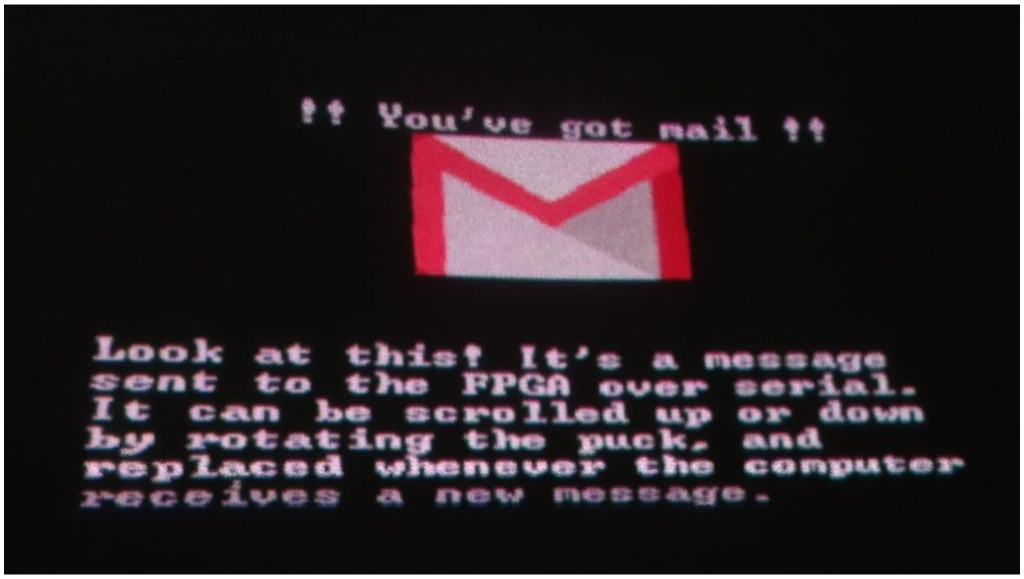
Program Switch:

This module switches between the 4 programs which can be fed into the keystone corrector module. It uses the CV click data as an input to determine when to switch to the next program. It also passes on the tilt data to the programs that need it.

Notification Display Program:

This interface is the main purpose of the device. It does this by taking ASCII characters from a computer sent over serial and storing them in strings to display on each line of text (see Appendix 6 for script to run on computer). It then displays them with a mail icon image over VGA, scrolled up and down by the puck's rotation from CV.

- **Display Gmail image:**
  A sprite of the gmail icon. Generated using imageconverter.py (Appendix 2), a script which takes a 256 color image and converts it to a verilog sprite.
- **Display ASCII text:**
  A 12-bit modification of cstringdisp.v from the 2005 course materials. For simple conversion of text to ASCII see appendix 7.



```
!! You've got mail !!

Look at this! It's a message
sent to the FPGA over serial.
It can be scrolled up or down
by rotating the puck, and
replaced whenever the computer
receives a new message.
```

Figure 16: The notification program displaying an incoming message

Display Spotlight:

Lights up a spotlight shaped region of the projector. Generated from an image using imageconverter.py (Appendix 2), a script which takes a 256 color image and converts it to a verilog sprite.

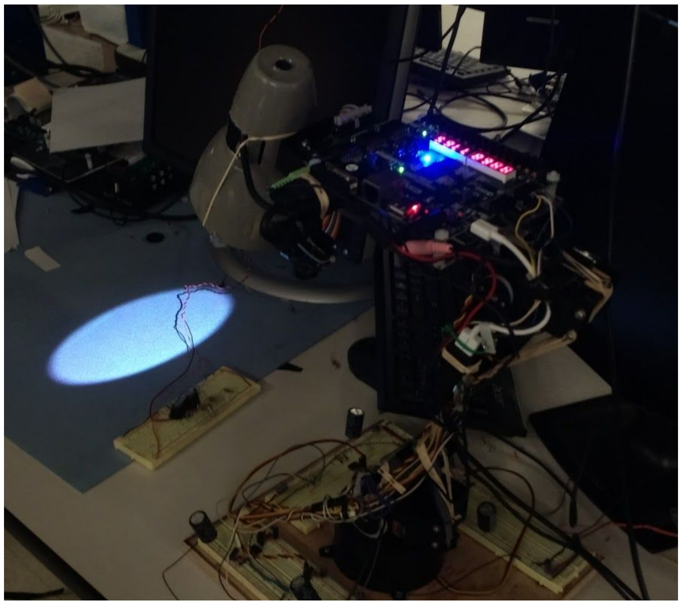


Figure 17: The robot arm displaying the spotlight module

Display Pong:

The pong program we used in lab 3, but rather than using a rectangle puck, it uses an amusing image found on the internet.

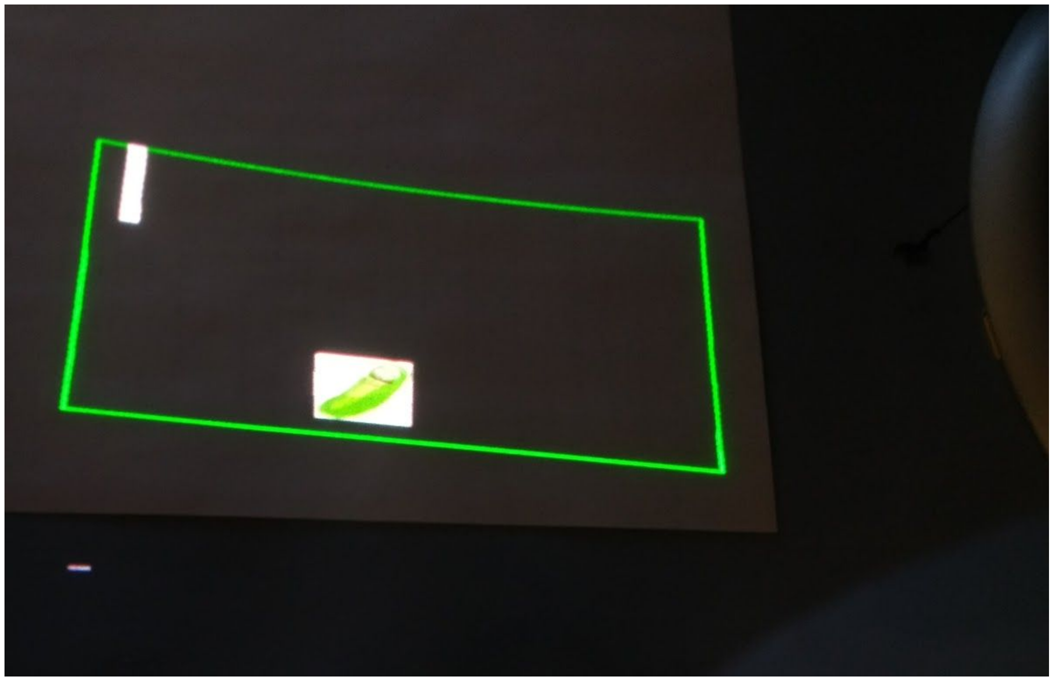


Figure 18: Playing keystone-corrected pong projected on a piece of paper

Display Dome image:

A 320x240 image of the dome on building 10. Generated to show off imageconverter.py (Appendix 2), a script which takes a 256 color image and converts it to a verilog sprite.

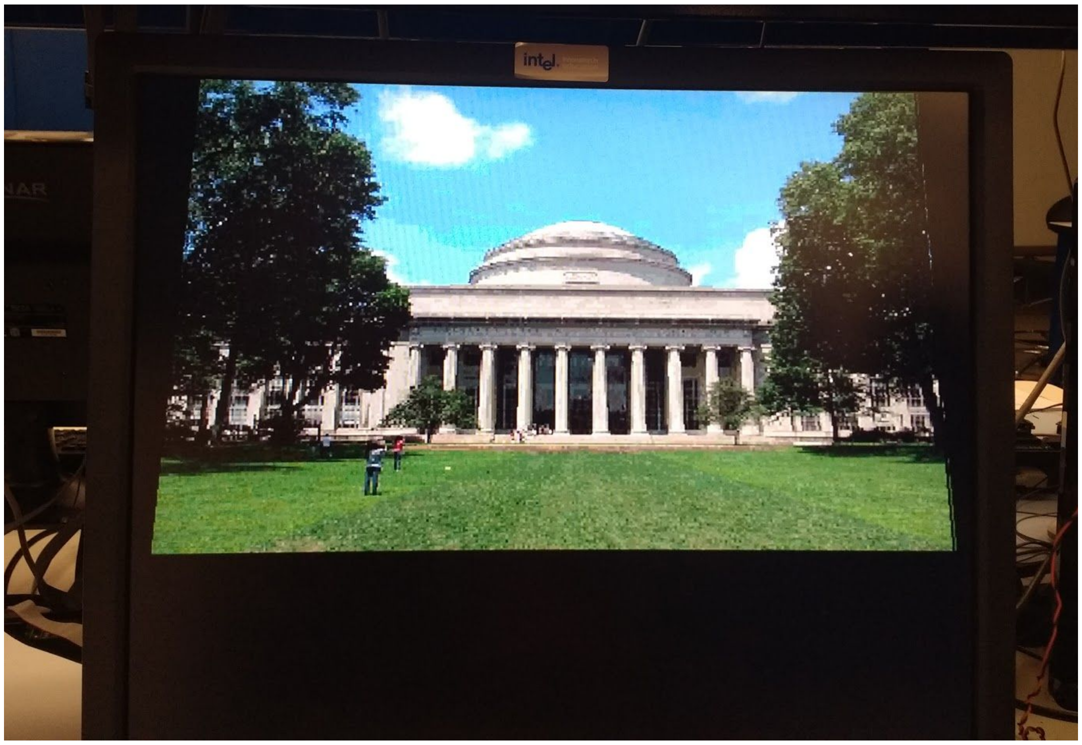


Figure 19: The dome displayed on the lab LCD to show more detail

RS232 to ASCII:

Takes an incoming serial connection from the computer over the Nexys 4 USB port and outputs received the most recently received byte. Luckily, Digilent provides an example project which contains VHDL to do almost exactly this using the built in usb to UART adapter. All that was changed here was the input clock and multipliers in the module to keep the same baud rate.

Sound Generator:

A combination of 3 modules. One creates a question-like sound, another a sad sound, and the last a neutral acknowledgement sound. Each module outputs a sound when triggered by the puck. The sound is sent to the 3.5mm audio jack of the Nexys 4, where it is played on attached speakers.

Appendix 1: Successive Angle Approximation MATLAB Script

```
% verilog successive approximation divider

screen_x = 640;
screen_y = 480;
num_bits = 10;

initial_x_res = ceil(log2(screen_x)); % 10
initial_y_res = ceil(log2(screen_y)); % 9

m_initial = logspace(-initial_x_res * log10(2), initial_y_res * log10(2), ...
    initial_x_res + initial_y_res + 1);

x = linspace(0,screen_x);
y = m_initial' * x;
figure; plot(x,y)
axis([0 screen_x 0 screen_y]); title('First pass results')
disp('Maximum angle error (first pass):')
disp(max(diff(180/pi*atan(m_initial))))
disp('Number of intervals (first pass):')
disp(length(m_initial) + 1)

m = m_initial(1);
for i = 1:length(m_initial)-1 % iterate over slopes
    % retrieve current slope
    current_slope = m_initial(i);

    % retrieve next slope
    next_slope = m_initial(i+1);

    % calculate remaining resolution
    remaining_res = num_bits - min(abs(log2([current_slope, next_slope])));

    % m_intermed = logspace(current_slope, next_slope, remaining resolution);
    m_intermed = logspace(log10(current_slope), log10(next_slope), ...
        remaining_res);
    m = [m m_intermed(2:end)];
end

y_new = m' * x;
figure; plot(x, y_new)
axis([0 screen_x 0 screen_y]); title('Second pass results')
disp('Maximum angle error (second pass):')
disp(max(diff(180/pi*atan(m))))
disp('Number of intervals (second pass):')
disp(length(m) + 1)
```

Appendix 2: Python image to 12 bit Verilog sprite generator, imageconverter.py

```
##########################################################################################
#                             12 bit Verilog sprite generator                             #
##########################################################################################
#                                                                                         #
# Converts an image into verilog for 12-bit VGA output                                    #
#                                                                                         #
# Requires PIL and Numpy                                                                  #
#                                                                                         #
# To conserve FPGA memory converting your image to 256 colors is highly recommended      #
#    -Use Gimp's index colors option finds the optimal palette of colors                  #
#    -Gimp can also scale the image to a lower resolution                                 #
#                                                                                         #
# Approximate resource usage of 320x240 256 color BMP image:                              #
#    -uses ~20% of the LUT on a Nexys 4 DDR (Artix-7 XC7A100T)                            #
#    -takes ~10 minutes to synthesize in Vivado                                           #
#                                                                                         #
##########################################################################################

from PIL import Image
import numpy


def run6111(filename):
    modulename = input('Verilog module name: ')
    SCALE = input('Scale pixels by 2^this_number (e.g 0 for each pixel 1x1, 1 for 2x2, 2 for 4x4,
etc):')
    print('Opening image'+filename+'...')
    intim = Image.open(filename)

    print('Converting image...')
    im = intim.convert('RGB')
    pixels = list(im.getdata())

    colors = []
    for pixel in pixels:
        r=str(hex(int((pixel[0])/16)))[2:]
        g=str(hex(int((pixel[1])/16)))[2:]
        b=str(hex(int((pixel[2])/16)))[2:]
        if (r,g,b) not in colors:
            colors.append((r,g,b))
    #print(colors)
    print('    '+str(len(colors)) + " colors found")

    print('Writing to file...')
    with open(modulename+"verilog.v", "w") as text_file:
        text_file.write('module '+modulename+'(input [10:0] x,hcount,input [9:0] y,vcount,output reg
[11:0] p_out);'+'\n\n\n')

        for color in colors:
            r=str(color[0])
            g=str(color[1])
            b=str(color[2])
            text_file.write('    parameter color'+r+g+b+" = 12'h"+r+g+b+';\n')
        text_file.write("    parameter CLEAR = 12'h000;\n")
        text_file.write('    parameter SCALE = '+str(SCALE)+';\n')
        text_file.write('    parameter HEIGHT = '+str(intim.size[1])+';\n')
        text_file.write('    parameter WIDTH = '+str(intim.size[0])+';\n\n\n')

        text_file.write('    reg [11:0] pixel;\n')

        locationwidth = int(numpy.log2(len(pixels)))
        text_file.write('    reg ['+str(locationwidth)+':0] location;\n')
        text_file.write('    always@(*) begin\n')
        text_file.write('        if ((hcount >= x && hcount < (x+(WIDTH<<SCALE))) && (vcount >= y &&
vcount < (y+(HEIGHT<<SCALE)))) p_out = pixel;\n')
        text_file.write('        else p_out = CLEAR;\n\n')
        text_file.write('        location = ((hcount-x)>>SCALE) + (((vcount -
y))>>SCALE)*WIDTH;\n\n')
        text_file.write('        case(location)\n')

        counter = 0
        for pixel in pixels:
            r=str(hex(int((pixel[0])/16)))[2:]
            g=str(hex(int((pixel[1])/16)))[2:]
            b=str(hex(int((pixel[2])/16)))[2:]
            text_file.write('            '+str(counter)+': pixel = color'+r+g+b+';\n')
            counter+=1
        text_file.write('            default: pixel = CLEAR;\n')
        text_file.write('        endcase\n')
        text_file.write('    end\n')
        text_file.write('endmodule\n')
        print('file "image.v" created')
def main():
    filename = input("Input image file name:")
    run6111(filename)


if __name__ == '__main__':
    main()
```

Appendix 3: Linear mapper for row compression

```
def linearmap(pixels_wide, number_to_skip):
    mapping = []

    pixels_before_skip = (pixels_wide/(number_to_skip))
    skiplist = []
    x = pixels_before_skip/2
    while (x < pixels_wide):
        skiplist.append(int(round(x)))
        x+=pixels_before_skip
    print(skiplist)
    print(len(skiplist), "elements skipped")

    x=1
    while x<=pixels_wide:
        if (x in skiplist):
            mapping.append(0)
            x+=2
        else:
            mapping.append(1)
            x+=1

    ans = ""
    for i in mapping:
        ans += str(i)
    return ans



with open("skiphlist.txt", "w") as text_file:
    for x in range(1,160): #for each possible value of x to scale within
        mapping = linearmap(320, x)
        length = len(mapping)
        #text_file.write("rowskip" + str(x) + " <= " + str(length) + "'b" + str(mapping) +
";\n")

        #if using a case statement: case(pixels_to_skip)
        text_file.write(str(x) + ": row <= " + str(length) + "'b" + str(mapping) + ";\n")
```

Appendix 4: Linear mapper for trapezoid angle

```
def linearmap(pixels_wide, number_to_skip):
    mapping = []

    pixels_before_skip = (pixels_wide/(number_to_skip))
    #print(pixels_before_skip)
    skiplist = []
    x = pixels_before_skip/2
    while (x < pixels_wide):
        skiplist.append(int(round(x)))
        x+=pixels_before_skip
    print(skiplist)
    print(len(skiplist), "elements skipped")

    x=1
    while x<=pixels_wide:
        if (x in skiplist):
            mapping.append(0)
            x+=1
        else:
            mapping.append(1)
            x+=1

    ans = ""
    for i in mapping:
        ans += str(i)
    return ans

with open("skipvlist.txt", "w") as text_file:
    for x in range(1,240): #for each possible value of x to scale within
        mapping = linearmap(480, x)
        length = len(mapping)
        #text_file.write("rowskip" + str(x) + " <= " + str(length) + "'b" + str(mapping) +
";\n")
        #if using a case statement: case(pixels_to_skip)
        text_file.write(str(x) + ": row <= " + str(length) + "'b" + str(mapping) + ";\n")
```

Appendix 5: Warp mapper for stretching bottom of the image

```
def warpmapper():
    # list of zeros if theta = 0
    anslist = []

    with open("affinelist.v", "w") as text_file:
        text_file.write('    case(theta);\n\n')

        for theta in range(0,45):
            text_file.write("        "+str(theta)+": affine <=
"+str(len(calculate_row_distances(theta)))+"'b"+str(calculate_row_distances(theta))+";\n")



def calculate_row_distances(theta):
    #A=((-0.001441*(theta**2))-(.004114*theta)+0.09448)*(10**-4)
    #entered wrong matlab, exaggerated warping though which looks cool
    A=((-5.391e-8*(theta**2))-(-3.863e-6*theta))#+-1.62e-6)
    #found through matlab based on measurements
    B=(0.0000253*theta**2)+(.001816*theta)+.20833333#0.2091 measured
    #print(str((A,B)))
    rowlist = []
    #squish_amount = theta/180 #skip every other pixel at 45 degrees
    squish_amount = theta/260 #determined by physical measurement
    for row_number in range(480):
        row_dist = round(((A*(row_number**2))+((B+squish_amount)*(row_number)))*4.8)
        rowlist.append(row_dist)
    print(rowlist)
    skiplist = []
    for x in range(0,479):
        if (rowlist[x] == rowlist[x+1]):
            skiplist.append(0)
        if ((rowlist[x]+1) == rowlist[x+1]):
            skiplist.append(1)
        if ((rowlist[x]+2) == rowlist[x+1]):
            skiplist.append(2)
        if ((rowlist[x]+3) == rowlist[x+1]):
            skiplist.append(3)

    ans = ""
    for i in skiplist:
        if i==0:
            ans += '00'
        if i==1:
            ans += '01'
        if i==2:
            ans += '10'
        if i==3:
            ans += '11'
    return ans

warpmapper()
```

Appendix 6: Send email to notification program

```python
import sys
import glob
import serial

#import smtplib
#import time
#import imaplib
#import email


def get_gmail():
    return input('type a message here: ')
    #replace with imaplib to get gmail or other email, removed for security


def string_to_lines(message):
    words = message.split()

    output = ''
    lines = []
    linenumber = 0
    currentline = ''
    for word in words:
        if len(currentline)+len(word) < 32:
            currentline+=word+' '
        else:
            lines.append(currentline)
            currentline = word+' '
    lines.append(currentline)
    for line in lines:
        output+=line.ljust(32)

    return output



if __name__ == '__main__':

    message = get_gmail()
    lines = (string_to_lines(message))


    ser = serial.Serial('/dev/ttyUSB4', 9600)  # open serial port
    print(ser.name)         # check which port was really used
    ser.write(lines)     # write a string


    ser.close()             # close port
```

Appendix 7: Text to verilog representation of ASCII converter

```python
text = input("Enter your text here: ")

ascii = ""
bits = 0
```

```python
print(len(text))

for letter in text:
    codestr = str(hex(ord(letter)))
    code = codestr[2:]
    ascii+=str(code)+'_'
    bits += 8
print('\n'+str(bits)+"'h"+ascii[:-1]+'\n\n')
```