Communication of Keyboard Input via on-Screen Pulsating Rectangle

Guthrie Cordone
ECE 6490
Dr. Richard Brooks
November 15th, 2016

1. Summary

In this project we develop a simple one-way communication protocol to allow an attacker to record a user's keyboard input by aiming a camera at a pulsating pixel on the computer monitor. We use python to convert user input into a serial ASCII bit stream that is displayed on the screen via a pulsating black rectangle. We then aim a webcam at the screen and perform change detection in the area of the pulsating rectangle to recover the ASCII character. In this paper we describe the development of our system and determine that it can reliably transmit ASCII characters at a rate of 0.5 per second. We leave a real life applicable covert implementation up for future work.

2. Introduction

An air-gapped computer network is a system that has been isolated from external networks, including adjacent local area networks and the internet. Air-gapping is something that is commonly done for computer systems that contain sensitive information or control critical systems, such as nuclear reactors. Data transmission between computers on an air-gapped network is intended to be done physically through human input or data transfer. However, recent research has shown that there are several ways that air-gapped computers can leak information to an outside network. [1] defines these types of exploits as out-of-band covert channels.

Several types of out-of-band covert channels have been shown to be possible, all of which rely on the abuse of computer peripherals that are unintended for communication purposes. Guri et al. [2] developed a method to communicate between computers using thermal manipulation and internal temperature sensors. Hanspach et al. [3] achieved communication between several laptops using high-frequency audio signals from their speakers. Other methods such as reflections from screens [4], IR communications [5], and vibrations [6] have been developed.

In this paper we focus on developing a simple visual communication channel that sends user input to a remote webcam via pixel pulsations on a computer monitor.

3. System Overview

Like any communication system we aim to have two major components: a transmitter and a receiver. In our system the transmitter will be the computer monitor of a compromised system that is running a program to convert all keyboard input into a pulsating pixel or group of pixels on the monitor. The receiver is a webcam or camera aimed at the compromised monitor that parses the pulsating rectangle into a serial bit stream and converts it to ASCII. Figure 1 shows a diagram of our proposed system.

In a real life covert scenario, the pulsating rectangle would be a pixel in the corner of the screen that is difficult for a user to see, and the camera would need to have a direct line of sight and a high enough zoom to be able to distinguish the flashing pixel. Since the hardware requirements for an actual implementation were beyond our reach, we designed a proof-of-concept implementation of our system using Python and Matlab that uses a standard webcam and a large pulsating rectangle.

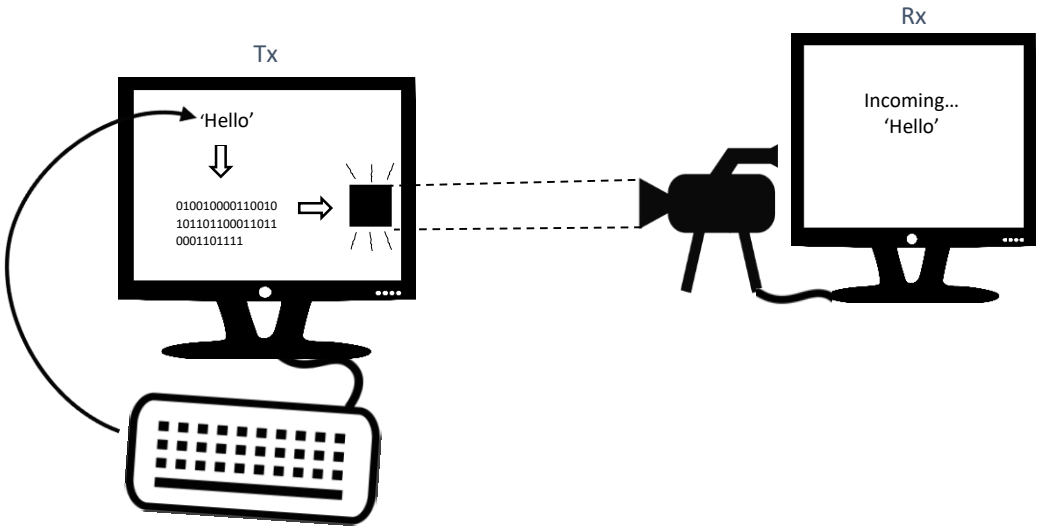

Figure 1: Diagram of our system

4. Transmission of Keyboard Input through Computer Monitor

We wrote the transmission component of our system using Python with the Pygame video game development module. The code for this component is given in the Appendix as user_input.py. We break down the transmission system into a sequence of three major steps:

1. Record user input
2. Convert user input into ASCII binary
3. Flash the ASCII binary to screen serially with a black rectangle representing a '1' and a blank screen representing a '0'

All three of these steps are executed within an infinite loop, allowing the pygame display to remain constantly open and the user to constantly be prompted for input.

4.1. Parsing User Input

For the first two steps, we simply use an input prompt with the raw_input() function and record the value to 'input_char' variable. Then, for each character in the input string, we use bin(ord(…)) to convert it to its respective ASCII binary value. The bin() function outputs a string of the form '0bXXXXXX' where 'X' is either a '1' or '0'. Since we are only interested in the 'X' values, we truncate the string to remove the first two characters. We also found that some ASCII inputs resulted in a 6 character binary string instead of the expected 7. For this case, we add a '0' to the beginning of the string. Once we have the 7 bit ASCII binary for each respective input, the system runs the loop to display the binary to screen.

4.2. Display serial ASCII stream

At the start of the program, we open a sufficiently large pygame window and set the background to white. Our goal is to communicate the current binary string by flashing a rectangle in the pygame

window. Whenever we want to send a '1', we flash a black rectangle for a specified time period, and whenever we want to send a '0', we turn off the rectangle for a specified time period. The rectangle flashing is called using the flash_rectangle() function and the time delay is specified using the time.sleep() function. In our loop to display an ASCII character, we first send a '1' before sending the ASCII binary. This first '1' triggers the receiver to begin reading the screen for the next 8 time periods. After sending the '1', we wait a time period and then send the ASCII binary serially through the flash_rectangle() function from most significant bit to least significant bit, waiting a time period between each flash. The time period wait is incredibly important because if the transmission sequence isn't in sync with the webcam sampling an incorrect ASCII sequence will get transmitted.

Once the program has finished sending the ASCII character, it sets the display white and waits for two time periods before repeating the process for the next ASCII binary string (if there is any). We found that if we wait for two time periods between the transmission of each character we eliminate a large amount of synchronization error caused by slight differences in the camera sample rate and the rectangle display rate. Figure 2 shows an example of our transmission system in the process of displaying a '1' with a black rectangle from the user input 'hello world!'.

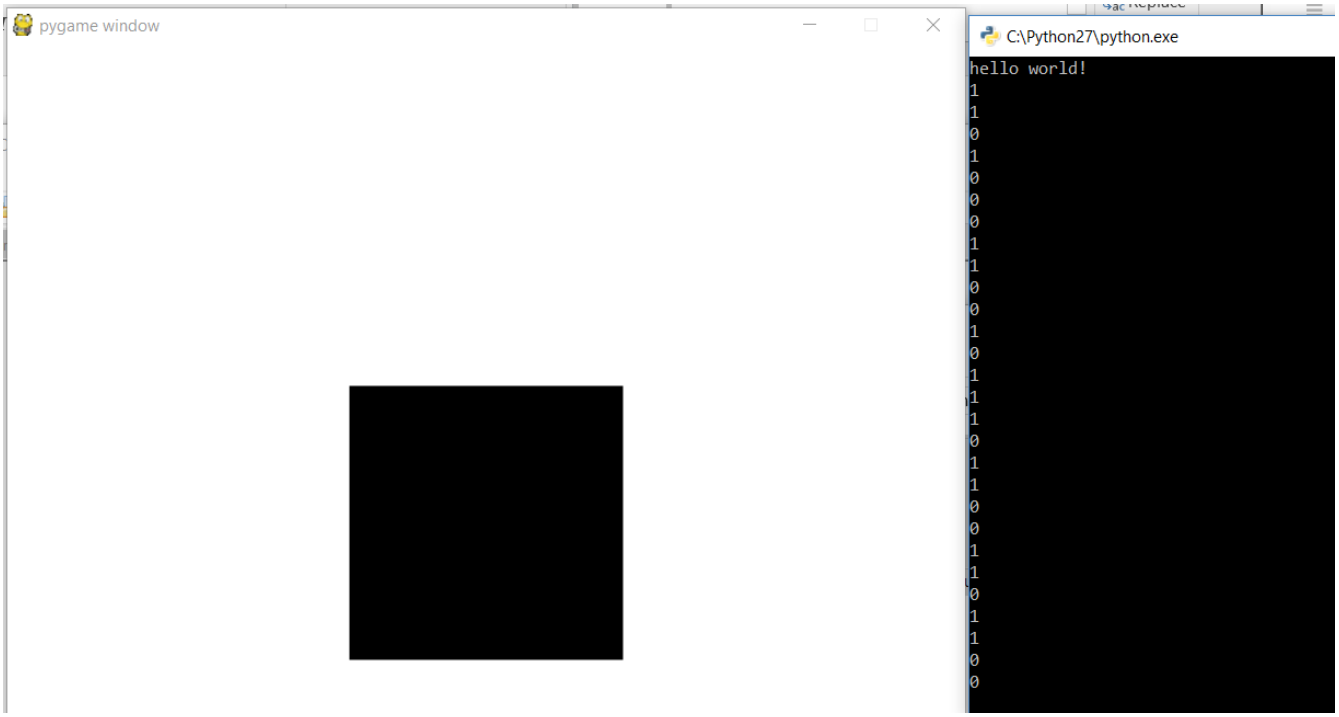

Figure 2: Example of user_input.py in the process of displaying the serial ascii bit stream of 'hello world!' via a pulsating rectangle using a pygame window

5. Converting Pulsating Rectangle back into ASCII via Webcam

The receiver component of our system uses a webcam to read the pulsating rectangle from the transmitter and convert it back into an ASCII character. We wrote this component in Matlab and the

code described in this section is given in the Appendix as cam_read.m and detect_change.m. The Webcam package for Matlab is required to run this code.

5.1. Image Processing and Change Detection

At the beginning of our code, we initialize a webcam object using the webcam() function and change its recording resolution to 160x120 (the smallest possible for our webcam). We choose the smallest possible resolution because we want each frame to process as quickly as possible. The code runs an infinite while loop in which a webcam frame is displayed each pass. We perform image processing on each frame and save the previous frame for use in change detection.

After taking a frame, we convert it into a binary array using the imbinarize() function. This function converts everything in an image to either black (represented by a '0') or white (represented by a '1') based on a threshold value. This allows simple change detection to be much faster. We also draw a rectangular aiming reticle in the middle of the frame before displaying for easy aiming of the webcam. Figure 3 shows an example output of our 160x120 binarized frame with a drawn on aiming reticle.

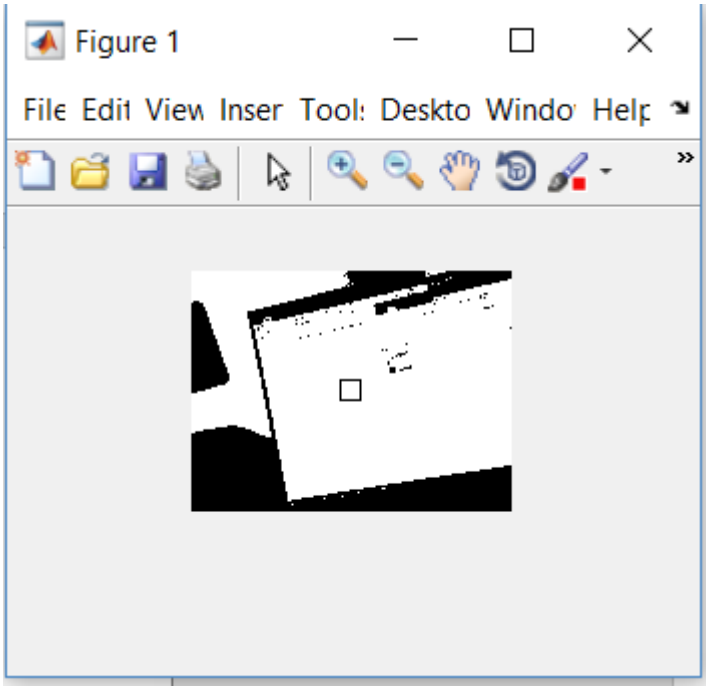

Figure 3: Binarized 160x120 webcam frame with aiming reticle drawn in the center

Our change detection algorithm is implemented in detect_change.m. This function takes in three arguments, a current binary image, a previous binary image, and a threshold value. The function computes the sum of all values in each frame and then subtracts the sum value of the old frame from the sum value of the new frame. If the magnitude of this difference is below the threshold value, then the function returns a 0, indicating that no change has occurred between the two input images. If the difference is negative and the magnitude is greater than or equal to the threshold value, then the new image has more black than the old image, and the function returns a 1. If the difference is positive and the magnitude is greater than or equal to the threshold value, then the new image has more white than the old image, and the function returns a -1.

For ease of use in our algorithm, we only send the portion of the image within the rectangular aiming reticle (as visible in Figure 3) through the change detection function. This allows us to aim where we want to perform the detection with the reticle.

5.2. State 1: Searching for Detection

After variable and webcam initialization, our receiver code runs in an infinite while loop. This loop functions with two different states as dictated by the 'detection' variable. The 'detection' variable is initialized to 0, signifying that we are searching for a detection within the aiming reticle shown in Figure 3. This subsection describes the operation of the system when 'detection' is set to 0. Note that for proper decoding, the webcam reticle must be aimed at the area of the screen that contains the pulsing rectangle from the transmission Python program.

When 'detection' is set to 0 an if statement runs the detect_change() function on the reticle area of the current webcam frame in comparison to the same area of the previous webcam frame. Since we know that the transmission part of our system (as described in Section 4) starts every transmission with a black rectangle pulse, we only trigger a detection when the output of the detect_change() function is 1. If the output is a 0 or -1, we determine that there is no detection and continue the recording loop. If the output of detect_change() is a 1, then the majority of the area inside the aiming reticle drawn on the webcam changed from white to black within the last frame, indicating that our the Python program has started to send an ASCII character. In this case, the 'detection' and 'take_sample' variables are set to 1 and the 'previous_bit' variable is set to 1. These changes will trigger the loop to enter its second state. Figure 4 shows an example transition that would trigger 'detection' to change to 1.

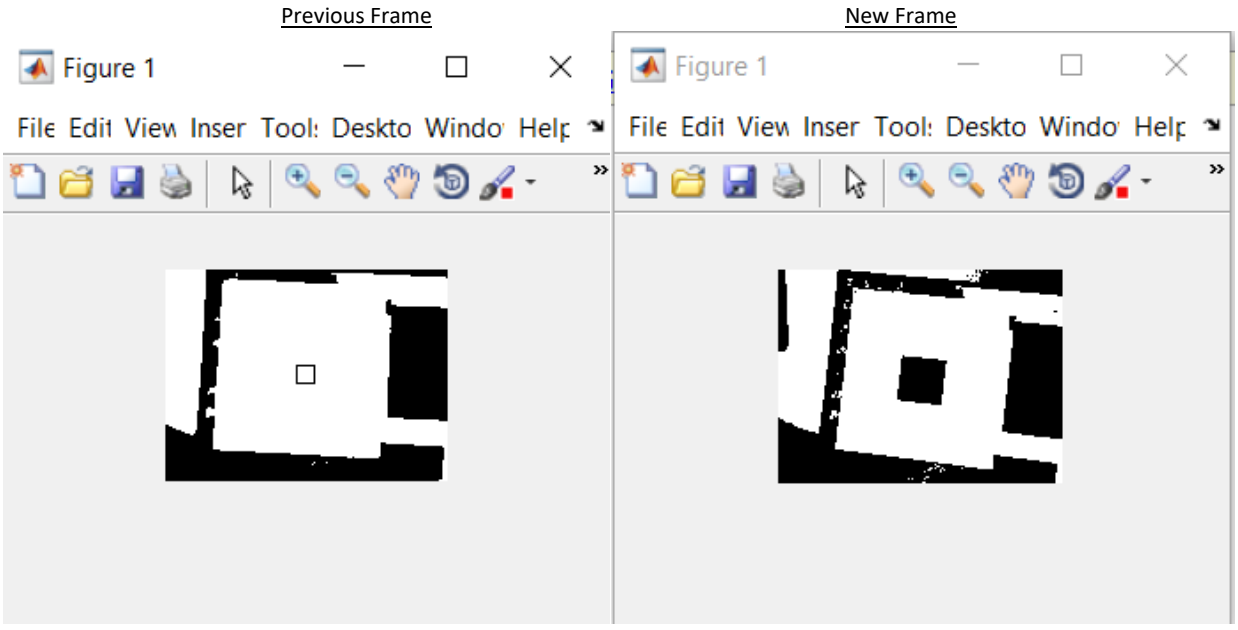

Figure 4: The black rectangle pulsing on screen triggers the sampling state to activate

5.3. State 2: Sampling

The second state of the loop is triggered after a white-to-black detection has occurred with the webcam aimed at the pulsating rectangle on the screen. This second state triggers when the variables 'detection' and 'take sample' are both set to 1. If both these variables are 1, then we begin to record the bit value of the current display. We first set 'sample_time' and 'take_sample' to 0, which resets the sample period. This is incredibly important because our receiver needs to only sample once per transmitted rectangle pulse to get a correct ASCII binary. To sample the current bit, we run the detect_change() function on the current webcam frame and compare it to the 'old_sample' frame. The 'old_sample' frame is the frame from the previous sample taken; if it is the first sample taken 'old_sample' is initialized as an all-black image. We use the results of the detect change function and the 'previous_bit' variable to determine whether to write a 1 or a 0 to the current location in the ASCII buffer. If the output of detect_change() is 0, then the rectangle is the same color as it was in the previous sample, and we write 'previous_bit' to the ASCII buffer. If the output of detect_change() is 1, then the rectangle changed from white to black over the last sample period and we write a 1 to the ASCII buffer and set 'previous_bit' to 1. If the output of detect_change() is -1, then the rectangle changed from black to white over the last sample period and we write a 0 to the ASCII buffer and set 'previous_bit' to 0.

Once a sample is taken, the loop continues to the end, where we have a conditional timer set up to activate only when 'detection' is set to 1. This timer simply adds the computation time of the current loop to 'sample_time' (which gets set to 0 when a sample is taken). The loop continues without sampling, adding the cumulative computation time to 'sample_time', until 'sample_time' is greater than or equal to our desired transmit period set in the Python display code. Once this occurs, we set take_sample to 1, allowing another sample image to be acquired for recording of the next bit in the ASCII buffer. We use a time-out system as opposed to a system pause to sync periods between our transmitter and receiver to allow us to display a live feed of the webcam without pausing the display while sampling.

The system remains in the sampling state until 8 samples have been taken (the ASCII buffer is full). At this point, we remove the initial transmission bit '1' from the ASCII buffer and convert the remaining bits into characters and display them to the screen. Once this happens we set 'detection' back to zero and go back to state 1. Figure 5 shows the result of our webcam receiver decoding the message 'Hello World!' sent by the pulsating rectangle Python code.

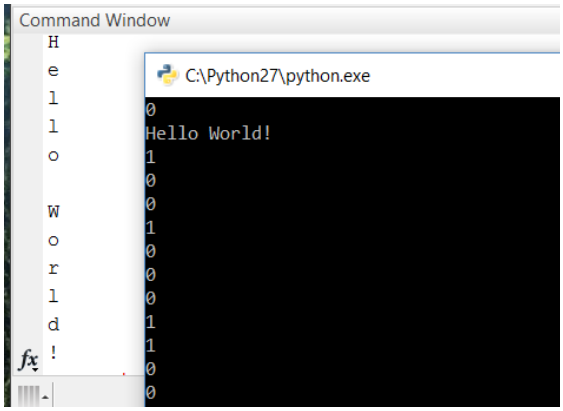

Figure 5: Our webcam receiver correctly reads the message 'Hello World!' sent over the monitor using a pulsating rectangle

6. Analysis and Discussion

One of the most important aspects of any communication system is it's data rate. Since our system requires use of a camera and a monitor, its data rate is capped by the refresh rate of the monitor and the maximum frame rate of the camera.

For our implementation, we aimed a 30fps Logitech HD webcam at a 144Hz refresh rate monitor for testing of our data throughput. The webcam was close enough to the monitor that the aiming reticle was completely filled by the pulsating rectangle (similar to the image in Figure 4). We tested the possible data rate of our system by sending a string of all possible keyboard characters into the transmission program. This test string is 93 characters long and seemed to be of sufficient length to highlight any transmission error of our system. The string is given as:

`1234567890-=qwertyuiop[]\asdfghjkl;'zxcvbnm,./!@#$%^&*()_+QWERTYUIOP{}|ASDFGHJKL:"ZXCVBNM<>?

We modified the display period in the transmission module and the receive period in the receive module to different values and recorded the number of transmission errors as well as their overall data throughput. We define the effective data rate in ASCII characters per second as the inverse of the 10 times the transmission period. The transmission period times 10 is the total time it takes to send one ASCII character (including the two period wait after completion of transmission). Therefore the effective rate is in units of ASCII characters per second, or message bytes per second. Table 1 shows our results for transmission periods of 0.20, 0.15, and 0.10 seconds.

The results of Table 1 show that with our current setup we should keep our transmission period above 1.5s for less than 10% character transmission error. We hypothesize that a drastic change occurs in transmission quality between 0.15s periods and 0.10s periods because these two periods bound the limit at which the receiver can remain in sync with the transmitter over the course of the 8 bit message.

We aren't accounting for the computation time or inherent delay taken to run the display portion of flash_rectangle() nor are we accounting for the time required to display the webcam visual on the monitor for the transmitter. These small un-accounted times will have a large impact on synchronization when we lower the display period to smaller levels, and it seems that the this critical period lies somewhere between 0.15 and 0.10 seconds of display time per period.

Table 1: Analysis of Transmission Error with Respect to Data Rate using 93 Character Input String

Transmission Period (s)	Effective Data Rate (B/s)	# Transmission Errors	Percent Error
0.20	0.50	2	2.15%
0.15	0.67	7	7.53%
0.10	1.00	75	80.64%

One thing that we could look into for future work is to encode our message using an error control code, such as a Reed Solomon or (11,7) Hamming Code. These codes would allow our system to correct a small number of bit transmission errors that occur through the system at the cost of longer codewords sent per ASCII character. Despite their ability to correct errors, however, a longer codeword sent may cause more errors since the sample time of the receiver is never completely in sync with the transmitter.

As such, the number of errors caused by a longer codeword may outnumber the number of errors correctable by using the longer codeword.

Another way to improve the error rate would be to investigate the wait time between character transmissions. Currently we wait two transmit periods between each ASCII character transmission. This wait time may need to be longer for smaller periods to allow the receiver to catch up in its looping.

7. Conclusion

In this project we developed a proof-of-concept out-of-band covert channel communication system that uses a Python code to display a user's keyboard input to the computer monitor using a pulsating rectangle and a Matlab code with a webcam to read the pulsating rectangle and convert it back into ASCII characters. We found that our system works with less than 10% transmission error for display periods above 1.5 seconds. At display periods below 1.5 seconds, our receiver becomes out of sync with the transmitter in the middle of each transmission, causing a large number of errors. In future work we would like to take into account computation times for both the transmitter and receiver to allow better synchronization between the two. We would also like to investigate the use of error correction coding and determine whether a longer codeword is worth it for minor error correction capability.

References

[1] Carrara, Brent. Air-Gap Covert Channels. Diss. Université d'Ottawa/University of Ottawa, 2016.

[2] Guri, Mordechai, et al. "BitWhisper: Covert signaling channel between air-gapped computers using thermal manipulations." 2015 IEEE 28th Computer Security Foundations Symposium. IEEE, 2015.

[3] Hanspach, Michael, and Michael Goetz. "On covert acoustical mesh networks in air." arXiv preprint arXiv:1406.1213 (2014).

[4] Backes, Michael, Markus Dürmuth, and Dominique Unruh. "Compromising reflections-or-how to read LCD monitors around the corner." 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, 2008.

[5] Balfanz, Dirk, et al. "Talking to Strangers: Authentication in Ad-Hoc Wireless Networks." NDSS. 2002.

[6] Subramanian, Venkatachalam, et al. "Examining the characteristics and implications of sensor side channels." 2013 IEEE International Conference on Communications (ICC). IEEE, 2013.

Appendix

user_input.py

```
import pygame
import time
import numpy as n

pygame.init()
black = (0, 0, 0)
white = (255, 255, 255)
size = (700, 700)
screen = pygame.display.set_mode(size)
screen.fill(white)
pygame.display.update()

display_per = 0.2

def flash_rect(bin_char):
	pygame.event.get()
	if bin_char == '0':
		screen.fill(white)
	if bin_char == '1':
		pygame.draw.rect(screen, black, (300,300,200,200), 0)
	pygame.display.flip()


while 1:
	for event in pygame.event.get():
		if event.type == pygame.QUIT: sys.exit()

	input_char = raw_input('')

	#convert input chars into serial ascii stream
	for ind_x in range (0,len(input_char)):
		input_bin = bin(ord(input_char[ind_x]))
		input_bin = input_bin[2:len(input_bin)]

		#make sure the ascii sequence is 7 bits long
		if len(input_bin) == 6:
			input_bin = '0' + input_bin

		#start every ascii sequence with a rect flash
		flash_rect('1')
		time.sleep(display_per)  #display period

		#flash ascii sequence
		for ind_y in range (0,len(input_bin)):
			flash_rect(input_bin[ind_y])
			print input_bin[ind_y]
			time.sleep(display_per)  #display period
		#wait for resync
		flash_rect('0')
		time.sleep(display_per*2)
```

cam_read.m

```
clear, clc

max_frame_time = 0;

%open webcam object and define resolution to use
cam = webcam(1);
res = [160 120];  %[ width  height ] =
cam.Resolution = sprintf('%dx%d',res(1),res(2));

%define change detection area
transmit_period = 0.2;  %clock period of data transmission
```

```
reticle_size = 10;  %size of square in center of webcam view
h_bound = res(1)/2-reticle_size/2:res(1)/2+reticle_size/2;
v_bound = res(2)/2-reticle_size/2:res(2)/2+reticle_size/2;

%record screen with webcam and perform detection
detection = 0;
take_sample = 0;
ascii_ind = 1;
frame_time = zeros(1,10);
ascii_buff = zeros(1,8);
old_frame = zeros(res(2),res(1));
old_sample = zeros(res(2),res(1));

while 1
    tic

    %take image and convert to black and white
    new_frame = snapshot(cam);
    new_frame = rgb2gray(new_frame);
    new_frame = imbinarize(new_frame);

    %show image with reticle
    show_frame = new_frame;
    show_frame(v_bound,res(1)/2-reticle_size/2) = 0; %left
    show_frame(v_bound,res(1)/2+reticle_size/2) = 0; %right
    show_frame(res(2)/2-reticle_size/2,h_bound) = 0; %bottom
    show_frame(res(2)/2+reticle_size/2,h_bound) = 0; %top
    if take_sample == 1 %show when sample occurs on visual
        show_frame(res(2)/2+reticle_size/2+10,h_bound) = 0; %top
    end
    imshow(show_frame);

    if ~detection  %search for start bit transmission
        change_val = detect_change(new_frame(v_bound,h_bound),old_frame,4);
        if change_val == 1 %contents of reticle go from white to black
            detection = 1;  %run ascii parse sequence
            take_sample = 1;
%             sample_time = 0;
            previous_bit = 1;
        end
    end

    if detection && take_sample  %parse ascii bits until end sequence recorded
        sample_time = 0;
        take_sample = 0;

        %parse ascii stream
        change_val = detect_change(new_frame(v_bound,h_bound),old_sample,4);
        if change_val == 0
            ascii_buff(ascii_ind) = previous_bit;
        end
        if change_val == 1
            ascii_buff(ascii_ind) = 1;
            previous_bit = 1;
        end
        if change_val == -1
            ascii_buff(ascii_ind) = 0;
            previous_bit = 0;
        end

        old_sample = new_frame(v_bound,h_bound);
%         sample_mat{ascii_ind} = old_sample;

        ascii_ind = ascii_ind+1;
```

```
        %end parsing sequence and print out character when buffer is full
        if ascii_ind == 9
            detection = 0;
            ascii_ind = 1;
            ascii_char = char(bin2dec(int2str(ascii_buff(2:8))));
            fprintf('%s\n',ascii_char)
        end
    end

    old_frame = new_frame(v_bound,h_bound);

    frame_time = toc;
    if detection
        sample_time = sample_time+frame_time;
        if (sample_time >= transmit_period)
            take_sample = 1;
        end
    end
end
```

detect_change.m

```
function change_val = detect_change(new_bwimg,old_bwimg,thresh)
%Returns a 0 if <=thresh pixels change between the old and new
%frames.  Returns a 1 if >thresh pixals change to black between the old and
%new frames.  Returns a -1 if >thresh pixels change to white between the old
%and new frames.

    bin_sum_new = sum(sum(new_bwimg));
    bin_sum_old = sum(sum(old_bwimg));

    %bit_diff < 0 if new frame has more black than old frame
    %bit_diff > 0 if old frame has more black than new frame
    %bit_diff = 0 if new frame and old frame have the same amount of black
    bit_diff = bin_sum_new-bin_sum_old;
    if abs(bit_diff) >= thresh
        if bit_diff < 0     %new frame has more black
            change_val = 1;
        end
        if bit_diff > 0     %new frame has more white
            change_val = -1;
        end
    else
        change_val = 0;     %new frame same as old frame
    end
end
```