Cannon by KFCams

Fahad Khan, Daniel Felix-Kim

EECS 395 Engineering System Design II
Ilya Mikhelson
Spring Quarter 2017

Table of Contents

Abstract **3**

Introduction **3**

Design Description **4**

- System Overview 4
- Block Diagrams 5
- Webcam Algorithms and Code 6
- Server Configuration 11
- Face Detection Algorithms and Code 13
- Website 15
- PCB Design 17
- 3D Printing 19

Final Product **20**

- Initial Goal vs. Final Product 20
- Performance and Limitations 20

Challenges Encountered **21**

Planning and Organization **22**

- Gantt Chart 22
- Communications Among Team Members 22
- Splitting Tasks Among Team Members 22

Conclusion - Daniel **23**

References **24**

Code **24**

Class Feedback - Daniel **24**

Abstract

The internet of things is an architecture for extending the functionality of small electronic devices with limited computational power. Within this type of architecture, small computing devices are networked to provide them with more powerful, remote computers. Thus, the endpoints are generally only responsible for data collection, transfer, and actuation. The remote servers (the “cloud”) are responsible for the heavier task of data processing and action planning. In this report we utilize the the internet of things architecture to create a cloud connected webcam. By leveraging the added computational power of the cloud, our webcam is able to automatically detect faces in an image frame, and track the face as it moves around.

As mentioned above, due to the limited computational power available on the webcam, we elect to offload the computation required for face tracking onto a remote server. Our selected architecture consists of a webcam as an endpoint that simply streams JPEG images to a remote server (over HTTP/TCP) and is placed on a pan and tilt based composed of two servo motors. The pan and tilt base allows the webcam to follow a face as it begins to exit an image frame using commands from a remote server. The webcam is a custom designed embedded system running compiled C on an Atmel microcontroller and is enclosed in a 3d printed case.

The server, running on an Amazon EC2 instance, is responsible for image processing, performing positioning calculations, and hosting the website files. For image processing, we utilize Python openCV libraries to perform face tracking with the Haar Cascades algorithm. Positioning calculations are then performed using a basic heuristic approach on the server and are translated to servo control commands on the webcam. A website is developed to display the stream while also allowing manual control of the webcam and its features through the use of websockets{1}.

With our architecture we were able to develop a webcam that can be remotely controlled and can automatically track face while streaming at roughly 3-4 frames per second. We also extended the face detection functionality to overlay Snapchat like filters on detected faces using bit masking in openCV. .

Introduction

The internet of things (IoT) is defined as the inter-networking of physical devices embedded with computational power. It allows several devices with limited computational power to leverage their connectivity to offload computational tasks. This allows a network of simple “smart” devices to perform complex tasks that may require running advanced machine learning algorithms and techniques on large data sets. These data sets are generated by the variety of sensors found on smart devices. Take, for example, a wireless webcam which generates large data sets corresponding to images. Image processing is usually too complex of a task to perform on the small microcontrollers often found on embedded webcams. Our embedded webcam uses an Atmel ATSAM4S8B capable of operation up to 120MHz with 512KB of flash memory and 128KB of SRAM. While these specifications are sufficient for interfacing with a camera and WiFi card, the addition of image processing libraries and algorithms would drastically cut down the frame rate. To accomplish our goal of implementing face tracking functionality, it was essential to have a significantly faster frame rate compared to last quarter’s design. Therefore, to add functionality to an embedded webcam, offloading computation to remote servers was the natural choice.

This report presents a webcam which streams its images to a server running OpenCV libraries. The server is capable of near-instantaneously running both the Haar Cascades

algorithm for detecting faces and the code for calculating camera positioning such that the webcam is able to automatically track faces. Moving the image processing to a server allowed for a faster frame rate and consequently low latency positioning corrections. Based on these requirements we generate the following goals for functionality the webcam must provide:

1. **Face detection on a remote server**
    a. Images should be marked with an indicator of faces present
    b. Images and website should be rehosted on remote server
        i. Users should be able to access the webcam from anywhere in the world
2. **Webcam rotation/tilt based on face positioning in frame**
    a. This necessitates at least 1 frame per second for accurate tracking
3. **Manual control of webcam position from the website**

Design Description

System Overview

As mentioned above, our webcam uses a fairly traditional IoT architecture. The webcam consists of three main components: A WiFi chip to send data to a server, a camera sensor that captures the actual images, and a microcontroller that allows us to control the interactions between all the components. The webcam circuit is a fully embedded system soldered onto a custom designed PCB. The design also includes several header pins to monitor serial output, program the WiFi chip, program the MCU with an Atmel ICE unit, and send power and PWM signals to two servos. The webcam can associate to a WiFi network and begin streaming images to our remote server over a persistent TCP client. Communication between the server and webcam occurs over the HTTP protocol. The webcam will post images and receive servo commands as a response. Images are transferred at a rate of roughly 3-4 frames per second.

For the WiFi chip, we use the ACKme AMW004 which runs Zentri-OS. Zentri-OS provides kernel calls for TCP capabilities. The file system capabilities of the chip are ignored for our purposes. The camera sensor is an Omnivision OV2640, a 2mp full color camera with a SCCB interface for data control. Both the WiFi chip and camera are controlled by an Atmel ATSAM4S8B microcontroller. I2C (TWI) is used by the MCU to communicate with the camera, while UART is used to communicate with the WiFi chip.

Our remote server runs on the Amazon Web Services (AWS) platform as an EC2 instance. The EC2 instance is a networked virtual machine running Ubuntu 16.04. We open port 80 on the server to allow HTTP requests to be made to the server. Requests are handled by NGINX 1.12, which implements a REST-like API for distributing and forwarding HTTP requests (see figure 3). For this project, it is configured to accept requests for webpages (GET root), websocket connections (GET-WS /camsocket), and for image processing (POST /detect). Image processing requests made to NGINX (POST /detect) are proxied to a local Tornado server running Python and OpenCV. This script is responsible for detecting faces and calculating positioning commands for the pan and tilt base under the webcam. After processing, the resulting positioning commands are sent back to the webcam WiFi card and translated to PWM values. The Tornado server alerts clients (browsers) to the presence of a new, processed image through the data websocket connection. Upon this message, browsers are instructed to fetch the new image with client-side javascript code, which are then displayed on the website, KFCams.me.

The client-side javascript code initializes data websocket connections for clients wishing to view the image stream. Functionality is included for switching between auto and manual control of the pan/tilt and for the application of face filters. These settings are communicated over the data websocket connections. Under manual mode, the positioning calculations performed by the server are replaced by commands sent from browsers.

Block Diagrams

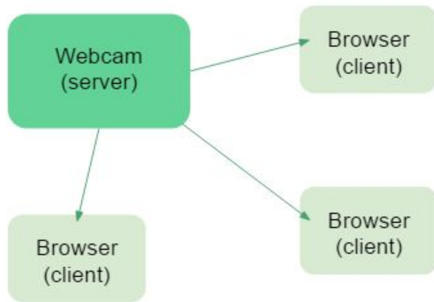


Fig 1: Webcam as server

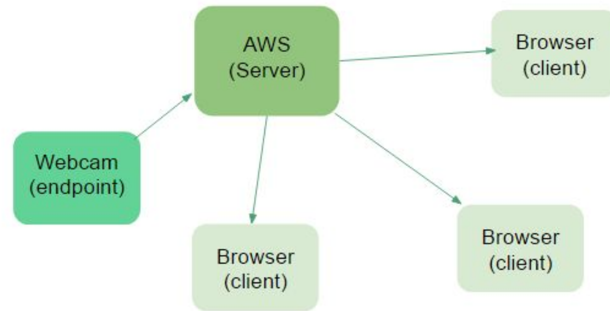


Fig 2: Webcam as endpoint

Figures 1 and 2 above present the two possible architectures for a wireless streaming webcam. Figure 1 treats the webcam as a server where clients (browser) connect directly to the WiFi chip. The WiFi chip stores the image and website files on its filesystem. The limitation of this architecture, as mentioned above, is that computational power is limited on the webcam microcontroller and on the WiFi chip. The microcontroller will not be able to perform face detection and the WiFi chip cannot handle more than roughly 4 simultaneous data streams (clients). Also, without further router configuration of port forwarding, the webcam can only be accessed by browsers on the same local network. The architecture in figure 2 solves most of these issues by using a virtual machine in the cloud to perform server duties.

Figure 3 below shows the details of the server architecture as described in the system overview. Note that currently, processed images are stored on the filesystem of the server.

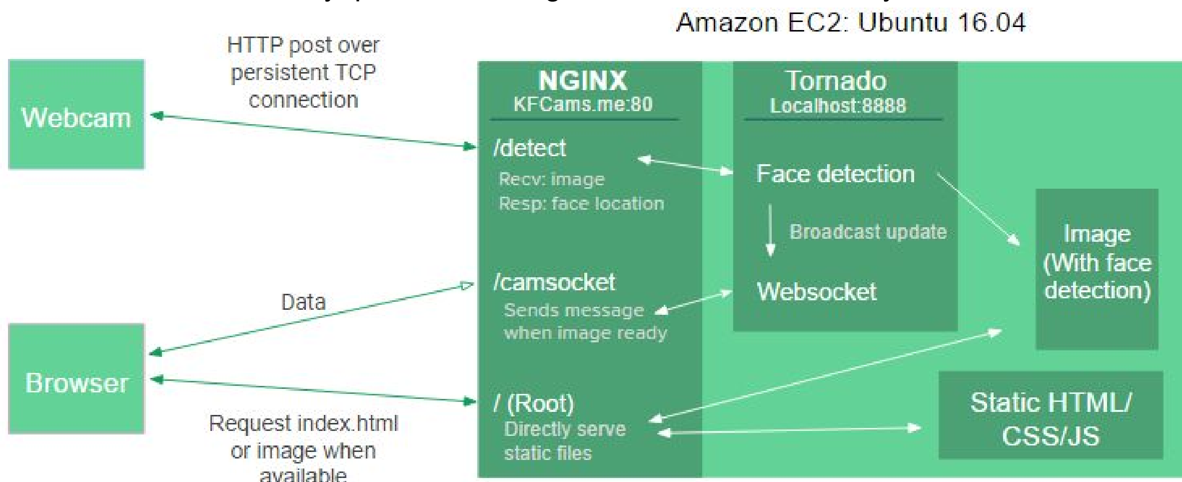


Fig. 3: Server Architecture

Webcam Algorithms and Code

The webcam was programmed in C using the Atmel Studio development environment. The structure of the program includes a main.c file that performs the actual execution of webcam operations. It is also responsible for setting up the PWM interrupt handlers for the pan and tilt servo motors, as well as the TCP client used to communicate with the server. Below is the skeleton for the main file with some code snippets.

```
int main (void)
{

	sysclk_init();
	board_init();
	wdt_disable(WDT);

	configure_usart_wifi();
	configure_wifi_comm_pin();
	configure_wifi_web_setup_pin();


	/* PLLA work at 96 Mhz */
	pmc_enable_pllack(7, 0x1, 1);

	//pwm setup  for servos
... Only channel settings are included ...

	pwm_clock_t clock_setting = {
		.ul_clka = PWM_FREQUENCY * PERIOD_VALUE,
		.ul_clkb = 0,
		.ul_mck = sysclk_get_cpu_hz()
	};

	pwm_init(PWM, &clock_setting);

	g_pwm_channel_led.alignment = PWM_ALIGN_LEFT;
	g_pwm_channel_led.polarity = PWM_HIGH;
	g_pwm_channel_led.ul_prescaler = PWM_CMR_CPRE_CLKA;
	g_pwm_channel_led.ul_period = PERIOD_VALUE;
	g_pwm_channel_led.ul_duty = INIT_DUTY_VALUE;
	g_pwm_channel_led.channel = PIN_PWM_LED0_CHANNEL;

	pan_duty = INIT_DUTY_VALUE;
	tilt_duty = INIT_DUTY_VALUE;

	/* Enable PWM channels for Servos */
	pwm_channel_enable(PWM, PIN_PWM_LED0_CHANNEL);
	pwm_channel_enable(PWM, PIN_PWM_LED1_CHANNEL);

	/* Wifi setup commands excluded */

...
```

```
        ack = 0;
        closed = 1;

        setup_flag = 0;

        //Check for network connectivity and setup option desired
        while (!ioport_get_pin_level(NETWORK_CONNECT_PIN)) {
                if (setup_flag) {
                        usart_write_line(BOARD_USART, "setup web\r\n");
                        setup_flag = 0;
                }
        }

        init_camera();
        configure_camera();

        ...

        //TCP Client initialization
        write_wifi_command("tcpc 52.15.48.211 80\r\n", 1);
        closed=0;
        ack=0;
        loop=0;

        while(1) {
                if (setup_flag)
                {
                        write_wifi_command("setup web\r\n", 1);
                        setup_flag = 0;
                }
                else if (ioport_get_pin_level(NETWORK_CONNECT_PIN)) {
                        ack = 0;
                        start_capture();
                        write_image_to_file();
                        set_pwm();

                }

        }
}
```

A wifi.c and wifi.h are included to take care of interactions with the WiFi card. This file includes some of the functions listed below. To approach these functions, we first referenced the USART hardware handshaking example to set up our own USART communication port. We then filled in the functions to handle interrupts by referencing the examples of interrupts in the OV7740_IMAGESENSOR_CAPTURE_EXAMPLE image sensor example program. We then filled in the functional programs as discussed below:

`void process_data_wifi (void)`: Processes the full command from the WiFi chip stored in the command buffer. This is done by using several if statements that call the strstr function to compare the command buffer to expected values. Since the servo positioning is calculated on

the server, it is received as a response by the WiFi chip. The response is parsed in this function. The structure is given below:

```
if (strstr(dump_buffer, "Set OK") || strstr(dump_buffer, "Success") ||
strstr(dump_buffer, "\nSuccess") || strstr(dump_buffer, "\nSet OK")) {
      ack = 1;

...

//Parsing the servo command response
else if(strstr(dump_buffer, "cmd=") || strstr(dump_buffer, "\ncmd=")) {
      post = 1;
      ack = 1;

      for (int i=0; i<BUFFER_SIZE; i++)
      {
            if (dump_buffer[i] == 'c' && dump_buffer[i+1] == 'm' &&
dump_buffer[i+2] == 'd')
            {
                  command[0] = dump_buffer[i+4];
                  command[1] = dump_buffer[i+5];
                  command[2] = dump_buffer[i+6];
                  command[3] = dump_buffer[i+7];

                  break;
            }
      }

}

//Check for closed TCP client
else if(strstr(dump_buffer, "Closed") != NULL || strstr(dump_buffer, "[Closed:
0]") || strstr(dump_buffer, "\n[Closed: 0]")) {
            ack = 1;
            closed = 1;
}
else if(strstr(dump_buffer, "not found") != NULL || strstr(dump_buffer,
"Unknown command") || strstr(dump_buffer, "\nUnknown command")) {
      ack = 1;
}

clear_dump();
```

`void write_wifi_command(char* comm, uint8_t cnt)`: Writes a command to the WiFi chip using `usart_write_line()`. Waits for acknowledgment or a timeout of cnt seconds. This is done by monitoring the global counts variable which is incremented every second (initialized in the main.c file).

`void write_image_to_file(void)`: Transfers images taken by camera to remote server. Note that we use HTTP 1.1 to communicate with the server so that we may reuse the TCP socket for multiple transfers.

```
void write_image_to_file(void)
{
	//Check if no image, return if so
	if (image_end == image_start || image_end < image_start) {
		return;
	}
	else {
		uint32_t num_dig = numPlaces(image_end-image_start);
		if (num_dig < 3) {
			return;
		}

		char create_string[30];
		char header[150];
		sprintf(create_string, "write 0 %lu\r\n",
83+num_dig+image_end-image_start);
		sprintf(header, "POST /detect HTTP/1.1\r\nContent-Length:
%d\r\nContent-Type: imagebin\r\nHost: 10.0.0.0\r\n\r\n",
image_end-image_start);

		if (closed)
		{
			open_connection();
		}

		pan_offset = 0;
		tilt_offset = 0;

		usart_write_line(BOARD_USART, create_string);

		//Send HTTP 1.1 header and image data
		usart_write_line(BOARD_USART, header);
		for (int jj = image_start; jj < image_end; jj++) {
			usart_putchar(BOARD_USART, cap_dest_buf[jj]);
		}

		counts = 0;
		while (counts < 1 && !ack) {}
		ack = 0;

		//Read response from server
		write_wifi_command("read 0 200\r\n", 1);
		ack = 0;
}
```

A camera.h and camera.c file are included for interacting with the OV2640 image sensor. The code here was approached in a similar fashion to wifi.c. We adapted examples from the OV7740_IMAGESENSOR_CAPTURE example program to write the functions for configuring the camera over TWI/I2C, starting an image capture and writing it to a buffer, and for enabling and handling the vsync interrupt. Adaption required switching to pclk1 and using the clock prescaler to bring plla down from 96 to 48 mhz. I2C is used in the case to write specific commands directly to the registers of the image sensor. Then we filled in the functional programs as seen below:

`uint8_t find_image_len(void):` Finds length of captured image to prepare for image transfer to WiFi chip. Note that we check 7 bytes of the JPEG header because we ran into several issues with corrupted JPEG images.

```
uint8_t start_flag = 0;
image_start = 0;
image_end = 0;

for (int ii = 0; ii < JPG_SIZE - 1; ii++) {
      uint8_t first_byte = cap_dest_buf[ii];
      uint8_t second_byte = cap_dest_buf[ii+1];
      ...
      uint8_t sixth_byte = cap_dest_buf[ii+5];
      uint8_t jfif_byte = cap_dest_buf[ii+6];

      if (!start_flag && first_byte == 0xFF && second_byte == 0xD8 &&
third_byte == 0xFF && fourth_byte == 0xE0 && fifth_byte == 0x00 && sixth_byte
== 0x10 && jfif_byte == 0x4a) {
            start_flag = 1;
            image_start = ii;
      }

      if (start_flag && first_byte == 0xFF && second_byte == 0xD9) {
            image_end = (ii+2);
            return 1;
      }
}
```

Finally, we include a servo.h and servo.c to handle the PWM interrupt and perform translations of positioning commands from the server to actual PWM values.

`Void PWM_Handler(void)` is responsible for updating the duty cycle value for pan and tilt servo motors. It also performs checks to ensure that we do not set the value too high or low.

```
void PWM_Handler(void)
{
      uint32_t events = pwm_channel_get_interrupt_status(PWM);

      /* Interrupt on PIN_PWM_LED0_CHANNEL */
      if ((events & (1 << PIN_PWM_LED0_CHANNEL)) ==
      (1 << PIN_PWM_LED0_CHANNEL)) {
            g_pwm_channel_led.channel = PIN_PWM_LED0_CHANNEL;

            pan_duty += pan_offset;
            pan_duty = (pan_duty < 800) ? 800 : pan_duty;
            pan_duty = (pan_duty > 2400) ? 2400 : pan_duty;

            pwm_channel_update_duty(PWM, &g_pwm_channel_led, pan_duty);
      }

... Repeat for channel 1 ...
```

```
}
```

`Void set_pwm(void)`**:** Performs the translation from the server positioning command to actual servo motor duty cycle values. The command is a simple bitmap that represents whether the camera should pan left or right, or if it should tilt up or down. The pwm frequency is set at 50Hz and the period is 20000 samples of which we offset the current duty cycle value by 2 samples at a time. Specific details of the bitmap can be seen in the code snippet below:

```
void set_pwm()
{
      pan_offset = 0;
      tilt_offset = 0;

      //bit 0 is pan enable
      if (command[0] == '1')
      {
            //bit 2 is left or right
            if (command[2] == '1')
                  pan_offset = OFFSET_FACTOR;
            else
                  pan_offset = OFFSET_FACTOR*-1;
      }

      //bit 1 is tilt enable
      if (command[1] == '1')
      {
            //bit 1 is up or down
            if (command[3] == '1')
                  tilt_offset = OFFSET_FACTOR;
            else
                  tilt_offset = OFFSET_FACTOR*-1;
      }

}
```

Server Configuration

As outlined in the system architecture, our backend system is composed of a publicly-facing server running NGINX and a processing server running Tornado. NGINX is a free, open-source, high-performance HTTP server and reverse proxy. We configure it to process HTTP requests according to the REST-like api defined above. It is directly responsible for serving static web files and images, but any requests for processing are forwarded to Tornado. The configuration is given below:

```
upstream tornadoserver {
      server 127.0.0.1:8888;
}
tcp_nodelay on;
client_body_buffer_size 50k;
```

```
server {
   listen 80 default_server;
   root /home/ubuntu/public_html;
   location /detect {
            proxy_pass http://tornadoserver/;
            proxy_redirect  http://localhost:8080/ /;
            proxy_read_timeout 60s;
            proxy_set_header          Host            $host;
            proxy_set_header          X-Real-IP       $remote_addr;
            proxy_set_header          X-Forwarded-For
$proxy_add_x_forwarded_for;
   }
      location /socket {
            # Forward to Tornado
            proxy_pass http://tornadoserver/socket;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
      }

   location /images {
      root /home/ubuntu/data;
   }
}
```

Processing is handled by a server running Tornado, a python web framework that we employ as a local server. NGINX forwards image processing requests to the root of this server while websocket connections are handled by /socket. Requests made to root are handled by the mainHandler class, which calls facedetect.py to perform face detection and save the image to the filesystem. If face filters are specified by clients, facedetectfilter.py is called so that filters can be applied to the image before saving. These files are discussed below.

Requests to the /socket are handled by the WebSocketHandler class. Methods implemented are as follows (implementations can be found at https://github.com/fahadkh/kfcams):

`check_origin(self, origin)`: Checks that websocket requests are made with from browsers connected to kfcams.me.

`open(self)`: Called when websocket connections are established. The current connection is added to the global list of clients that must be updated when new images arrive.

`on_message(self, message)`: Data messages from clients. This data is parsed and saved to be sent to the webcam when it makes requests. Data may include settings for auto versus manual track, filters on or off, or actual commands for panning and tilting the webcam.

`on_close(self)`: Called when clients disconnect. The function removes them from the global list of connected clients.

A global function called update_clients() is also included and called when the mainHandler is done processing an image. It tells connected clients that they may request the new image.

Face Detection Algorithms and Code

For the face detection processing, we used Haar Feature-based Cascade Classifiers as implemented in OpenCV. This algorithm is a machine learning based approach, where the model is trained using various images with and without faces. In our facedetect.py, we use the standard method of using pre-trained classifier XML files included in OpenCV to train a face and eye detector.

```
face_cascade =
cv2.CascadeClassifier('classifiers/haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('classifiers/haarcascade_eye.xml')
img1 = cv2.imread('face.jpg')
image_rows,image_cols,image_channels = img1.shape
gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray1, 1.3, 5)
```

For the standard facedetect.py with no filters, upon detecting the face, the function draws a blue rectangle around the face, and green rectangles around eyes. The center of the face is then calculated so that it can be used for tracking.

```
for (x,y,w,h) in faces:
	cv2.rectangle(img1,(x,y),(x+w,y+h),(255,0,0),2)
	centerFrame = x+w/2,y+h/2
	cv2.rectangle(img1,(centerFrame[0],centerFrame[1]),(centerFrame[0],center
Frame[1]),(0,0,255),2)
	roi_gray = gray1[y:y+h, x:x+w]
	roi_color = img1[y:y+h, x:x+w]
	eyes = eye_cascade.detectMultiScale(roi_gray)
	for (ex,ey,ew,eh) in eyes:
		cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)
```

We chose an area of the image where the face is considered centered, and if it went outside the boundaries, we sent a 4 digit binary number to the microcontroller. The command[0] and command[1] values directed whether the camera should pan or tilt respectively, with 1 confirming the motion. The command[2] and command[3] determined the direction that the camera should move.

```
	if (not centerFrame[0] == 0 and not centerFrame[1] == 0):
		if (centerFrame[0] < centerImage[0]-offsetImage[0]): #Face is to
the left of center
			command[0] = 1 #Pan Camera
			command[2] = 0 #Pan left
		elif (centerFrame[0] > centerImage[0] + offsetImage[0]): #Face is
right of center
			command[0] = 1 #Pan Camera
			command[2] = 1 #Pan right
		if (centerFrame[1] < centerImage[1] - offsetImage[1]): #Face is
below center
			command[1] = 1 #Tilt Camera
			command[3] = 1 #Tilt down
		elif (centerFrame[1] > centerImage[1] + offsetImage[1]):
			command[1] = 1 #Tile Camera
			command[3] = 0 #Tilt up
```

To implement the filters, we had to resize and center the filter.png image to properly fit over each face. The filters were applied using bitwise masking, which required the mask size to match the size of the region of interest in the original image. This process was outlined in the OpenCV Python Documentation and is similar to what Noah Dietrich did for adding mustaches to a video stream[2].

```
for (x,y,w,h) in faces:
	centerFrame = x+w/2,y+h/2
	roi_gray = gray1[y:y+h, x:x+w]
	roi_color = img1[y:y+h, x:x+w]
	eyes = eye_cascade.detectMultiScale(roi_gray)

	face_filter = cv2.imread('dog_filter.png')
	x1 = x - 15
	x2 = x + w
	y1 =  y -50
	y2 = h+y + 70

	rows,cols,channels = face_filter.shape

	# Check for clipping
	if x1 < 0:
		x1 = 0
	if y1 < 0:
		y1 = 0
	if x2 > image_cols:
		x2 = image_cols
	if y2 > image_rows:
		y2 = image_rows
```

```
        # Re-calculate the width and height of the filter image
    filterWidth = x2 - x1
    filterHeight = y2 - y1
    # Re-size the original image and the masks to the filter size

    # Now create a mask of logo and create its inverse mask also
    img2gray = cv2.cvtColor(face_filter,cv2.COLOR_BGR2GRAY)
    ret, mask = cv2.threshold(img2gray, 10, 255, cv2.THRESH_BINARY)
    mask_inv = cv2.bitwise_not(mask)

    filter_apply = cv2.resize(face_filter, (filterWidth,filterHeight),
interpolation = cv2.INTER_AREA)
    mask = cv2.resize(mask, (filterWidth,filterHeight), interpolation =
cv2.INTER_AREA)
    mask_inv = cv2.resize(mask_inv, (filterWidth,filterHeight), interpolation
= cv2.INTER_AREA)

    # Create region of interest (roi) in base image
    roi = img1[y1:y2,x1:x2]
    # Now black-out the area of logo in ROI
    img1_bg = cv2.bitwise_and(roi,roi,mask = mask_inv)
    # Take only region of logo from logo image.
    img2_fg = cv2.bitwise_and(filter_apply,filter_apply,mask = mask)
    # Put filter in ROI and modify the main image
    dst = cv2.add(img1_bg,img2_fg)
    img1[y1:y2,x1:x2] = dst
```

Website

The website is used by clients to observe the image stream from the webcam. It is also used to adjust webcam settings such as the tracking mode (manual versus auto) and whether faces should be outlined by a bounding box or with snapchat-like filters. The website layout is programmed using HTML, CSS, and bootstrap (a CSS framework that adds several CSS classes for element placement and design).

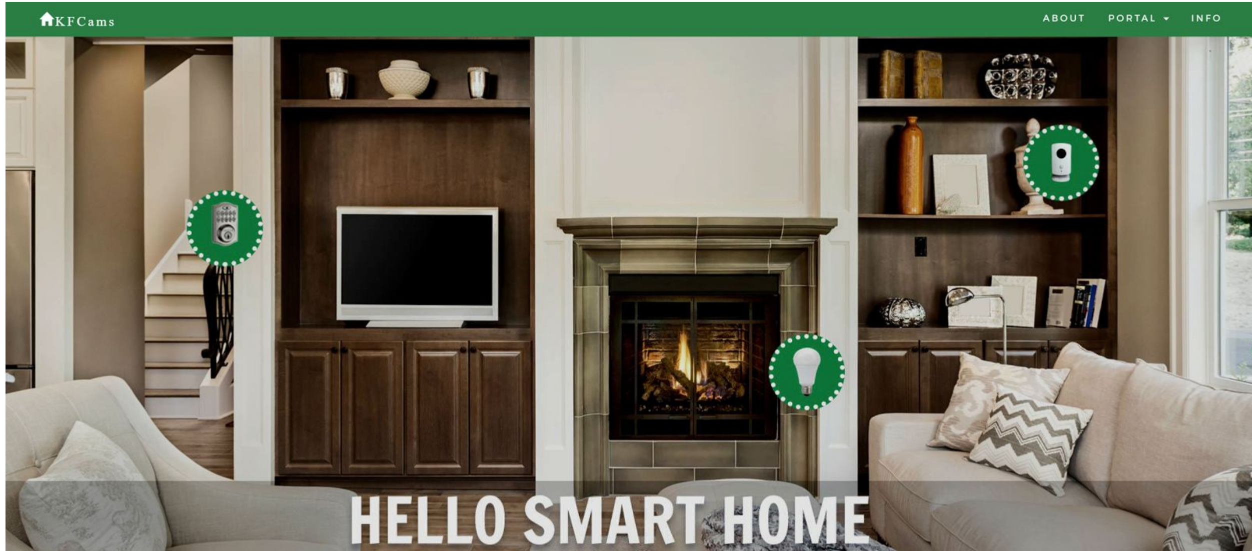


Fig 4. Homepage

The website is modeled after an entire home automation system, but currently only supports the webcam functions. To actually support the image stream, a client-side javascript file is included to enable the websocket data stream. This file maps the buttons seen in figure 5 below to messages sent over the websocket connection. The mapping is given in table 1. Note that a timestamp is included in the image request to ensure that the browser does not cache images. Up, Down, Left, and Right buttons are directly overlaid on the image and appear when the cursor hovers over them.

Table 1. Action and Result, webcam_functions.js
(https://github.com/fahadkh/kfcams/blob/master/public_html/scripts/webcam_functions.js)

Action	**Message/Result**
Button Press: Filter On	{fmode: “on”}, Filter On button becomes Filter Off
Button Press: Filter Off	{fmode: “off”}, Filter Off button becomes Filter On
Button Press: Manual Track	{mode: “manual”}, Manual Track button becomes Auto Track
Button Press: Auto Track	{mode: “auto”}, Auto Track button becomes Manual Track
Button Press: Stop Webcam	on_close() called, Websocket closed, Stop Webcam becomes Start Webcam button
Button Press: Start Webcam	on_open() called, Websocket opened, Start Webcam becomes Stop Webcam button
Button/Key Press: Up or W key	{command: “up”}
Button/Key Press: Left or A key	{command: “left”}
Button/Key Press: Down or S key	{command: “down”}

Button/Key Press: Right or D key	{command: “right”}
On message	GET "/images/img.png?time=" + new Date().getTime();

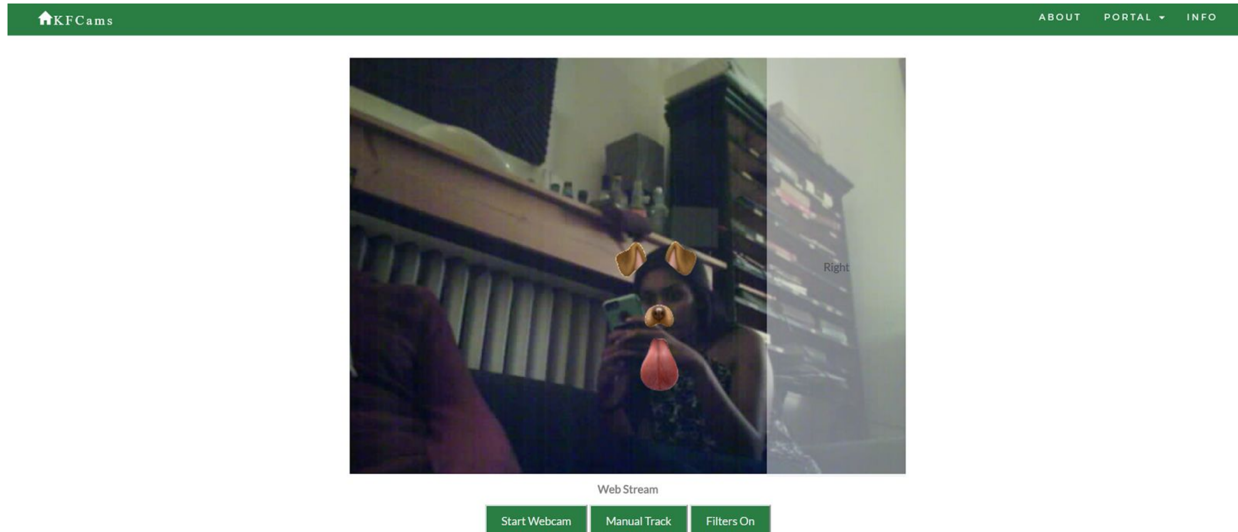


Fig. 5. Image stream page

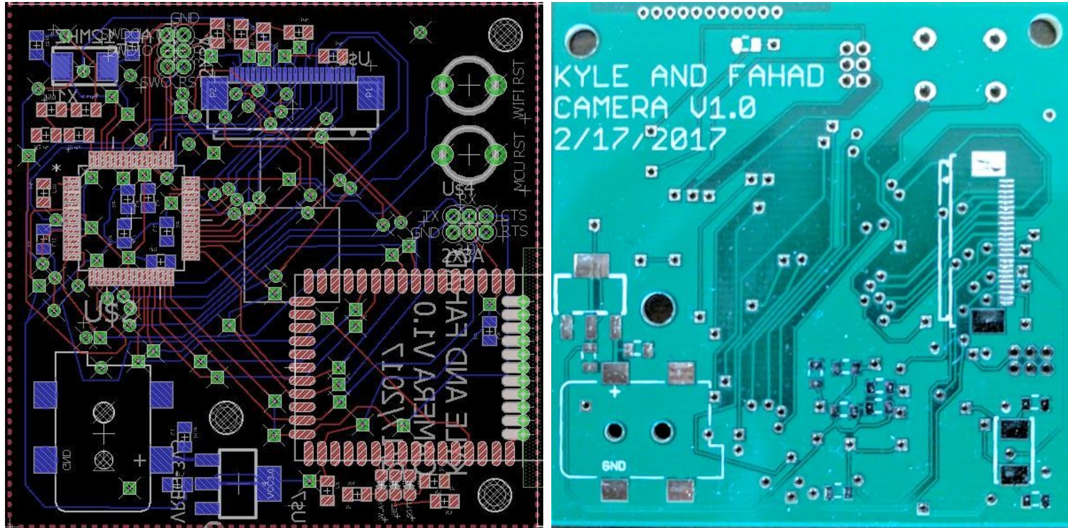
PCB Design

The final embedded system PCB is the result of 4 iterations. The first PCB was designed with a focus on reliability and easy debugging. The full 50mm x 50mm allowable board size was used to reduce complexity of routing and vias were included on most wires for easy testing with multimeters. To reduce thickness, all tall components (camera, headers and barrel jack) were placed on one side of the board, with the wifi card and MCU on the other side.

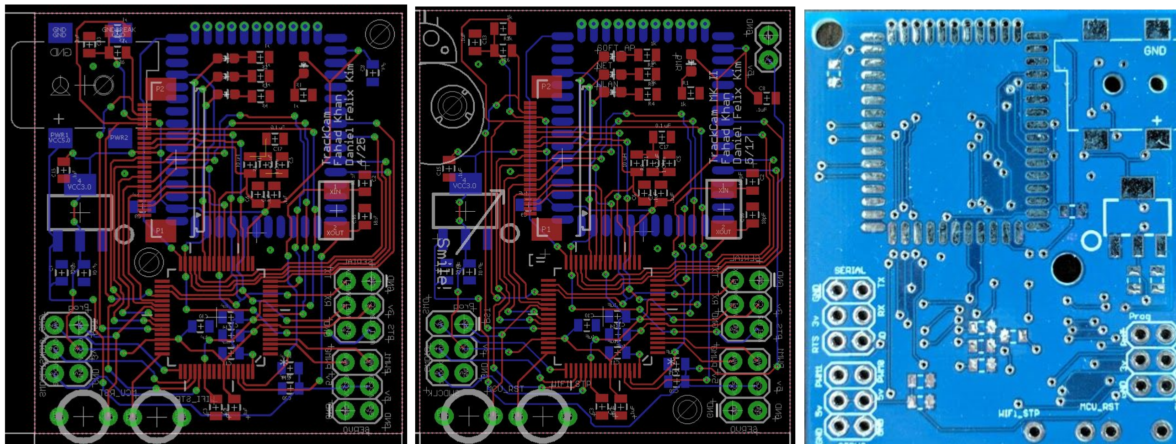
After validating the schematic, we iterated on the design to reduce the size of the board and add connections for servo motors. For this iteration, we focused on the best components placement to attain the smallest size board. We also adjusted component placement to account for the new pan and tilt servo motor base. The barrel jack was move to the bottom of the board so that the power wire could stick out from the bottom of the camera and run down the front of the base. This iteration had a board size of 43.82mm x 38.10mm.

A revision was made to the second iteration to fix a few missing connections to capacitors and to remove the barrel jack. The barrel jack was replaced by headers. These were attached to a barrel jack on the base of the webcam with thinner wires so that the barrel jack would not obstruct movement as much as it had with the second board iteration. A separate barrel jack to header breakout board was also designed. The silkscreen layer was improved with better labels for each header, button, and LED. The board size was unchanged.

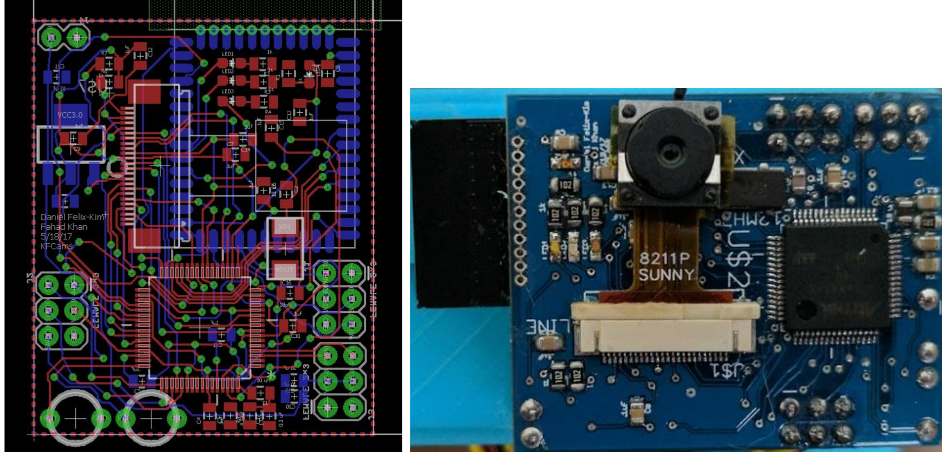
A final revision was made to drastically reduce the size of the board. We removed mounting holes and instead elected for a press fit for the board. Component placement remained roughly similar, but trace routing was redone completely to accommodate the smaller size. This iteration had a board size of 41mm x 33.5mm.



Version 1



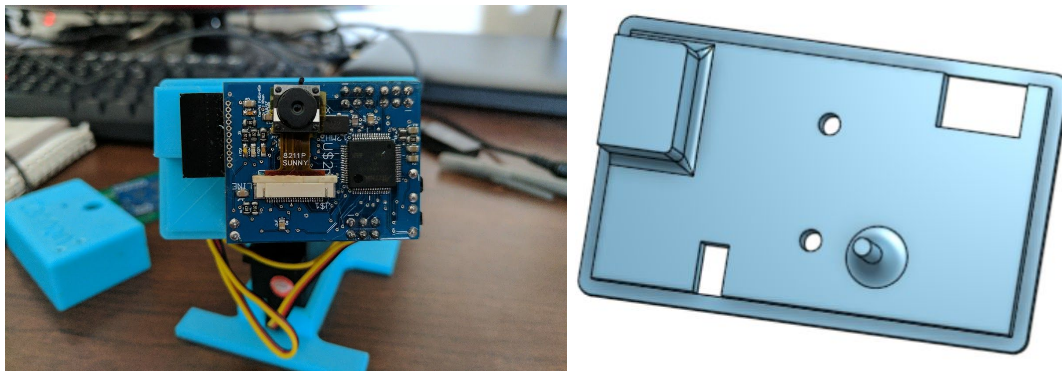
Version 2 and 3



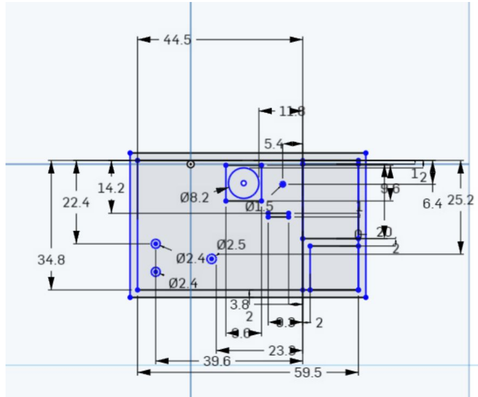
Version 4 (final and soldered)

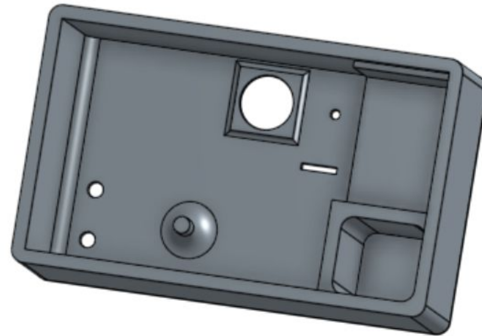
3D Printing

The enclosure for the webcam consists of two parts. The actual webcam is enclosed in a rectangular case designed to center the camera sensor on the pan and tilt base. Holes were included for the camera, power and status LEDs, reset and setup buttons, and for connections to headers. Since no mounting holes were included on the actual PCB, we designed the case to clamp down on the board. Since the webcam is designed to be placed on a desk and not be mobile, we found this satisfactory in terms of retainment. The back lid of the enclosure includes mounting holes to attach to the pan and tilt base. This lid is first attached to the mounting points, then the PCB is placed onto the lid and wires plugged in. The front cover is then snapped onto the lid. The second part of the enclosure houses the base of the pan and tilt motors to stabilize them and prevent tipping. A slot for the barrel jack is also included.



Enclosure lid and base mount, PCB attached





Enclosure cover with schematic

Final Product

Initial Goal vs. Final Product

As mentioned in the introduction, the goals for the project were as follows:

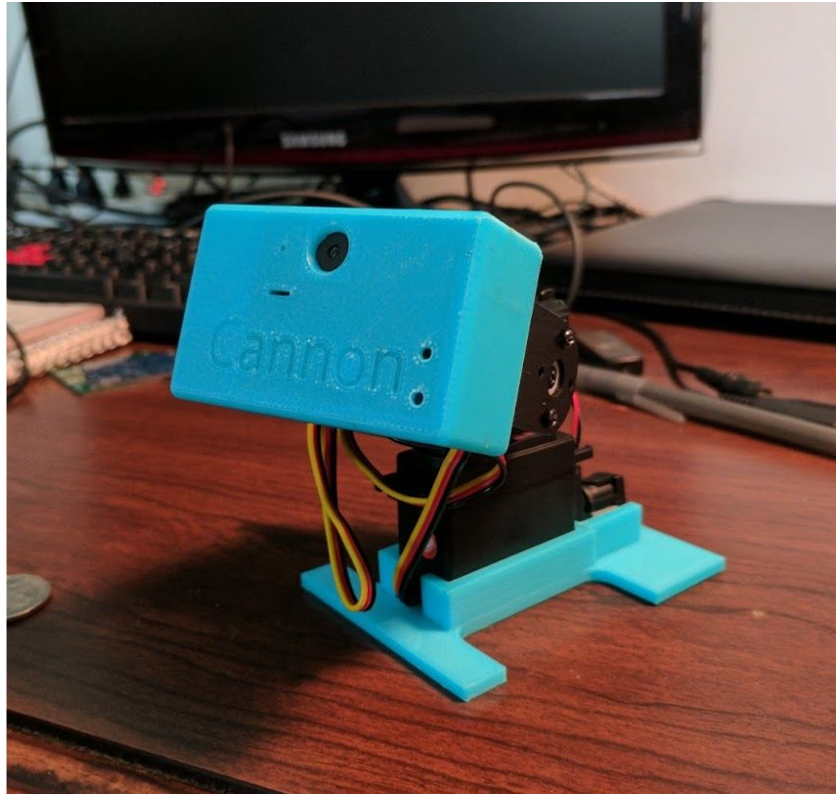
1. **Face detection on a remote server**
    a. Images should be marked with an indicator of faces present
    b. Images and website should be rehosted on remote server
        i. Users should be able to access the webcam from anywhere in the world
2. **Webcam rotation/tilt based on face positioning in frame**
    a. This necessitates at least 1 frame per second for accurate tracking
3. **Manual control of webcam position from the website**

As highlighted above, we were able to hit all of our goals for the project. The Python programs used Haar Cascades and bitwise masking for face detection and adding filters. They also kept track of the face's position relative to the center, which was used to determine whether the servos should be actuated. The website allows for manual control of the webcam using onscreen buttons and also present filter options.

Performance and Limitations

The webcam is able to stream 480p images consistently at around 3-4 frames per second, provided a strong internet connection. However, one recurring issue is that the image

becomes corrupted and stops the image stream for several seconds before resetting and functioning normally. This happened more often with poor WiFi connections. Improving frame rate may not be possible with netcode alone. We may also require a fast frame buffer that can take the processing load off of the microcontroller.



Final Product

Challenges Encountered

We struggled a lot initially with configuring the server to run all of our locally prototyped python code. The bulk of our face detection required the OpenCV library for Python. This library requires its own dependencies which we also had to install and configure. Eventually we had to compile the OpenCV library from source, install it, and link it to Python.

We also faced some of the real-world challenges of hosting a public server. The internet is being consistently port scanned for open ports. Since our server is open on port 80, several automated attempts have been made to conduct path traversal attacks. These attacks attempt to traverse the server filesystem from our index.html file to also get to well known configuration files. For example, an attacker’s automatic scanner might try to GET a poorly placed PHP admin file to see if a path traversal attack is possible. Success would indicate a potentially vulnerable site, prompting further manual investigation. However, with default NGINX configuration and smart placement of web files, we were able to mitigate the impact of these scanners.

On a hardware level, setting up the PWM control was challenging since it required us to look at all of the interrupts and understand the sometimes confusing variable names in Atmel code examples. For example, the variable referring to the PWM period value was actually the number of samples per cycle.

Regarding the PCB, debugging was difficult when we faced issues such as shorts or airwires that Eagle missed. We were able to isolate the problems by looking at the board file while also ensuring that the different power connections had the proper voltages. Luckily, the only missed connection was in between two adjacent capacitors, so we were able to solder them together. Soldering the board was slightly more difficult with the final compact version, but we were able finish the first attempt. Eagle 8 also created some issues such as inconsistent DRC's, though we were able to resolve this by having each of us check the board.

Planning and Organization

Gantt Chart

	Start Date	End Date	Timeline	Status
EECS 395/495	Mar-28	Jun-5		
Define project goals	Mar-30	Apr-2		Complete
Research components	Mar-30	Apr-4		Complete
Prepare order	Mar-30	Apr-4		Complete
Create Motor Breakout	Mar-31	Apr-6		Complete
Figure out script hosting on AWS	Apr-5	Apr-18		Complete
Modify MCU code to POST image to server	Apr-17	Apr-21		Complete
Update webcam board to include servo connector	Apr-20	Apr-27		Complete
Prototype python openCV locally	Apr-17	May-3		Complete
Add POST response with object positioning to server code	Apr-27	May-10		Complete
Rehost website on server	May-4	May-12		Complete
Add image overlay for detected objects (Server)	May-3	May-19		Complete
Prototype motor control code	Apr-24	May-15		Complete
Complete final embedded system board	May-8	May-18		Complete
Modify mcu code to respond to object position	May-15	May-26		Complete
Solder final embedded system, resolve issues	May-26	Jun-5		Complete
Final Presentation	May-30	Jun-5		Complete
		Burndown		

Communications Among Team Members

We were in constant communication through electronic means. We would have a meeting at least once a week on Wednesdays. Since both of our schedules were not very rigid, we did not have a standard meeting time but would be sure to meet at least once to start on the main element of a task and then further split the task for independent completion.

Splitting Tasks Among Team Members

Since we laid out all our tasks on the Gantt Chart, we were able to split responsibilities line by line. However, for the most part, we would meet in person to start work on a task and only split the smaller subtasks if needed. We handled much of the server configuration early on and focused on hardware later in the quarter. Server communication and configuration was led

by Fahad while face detection and filters were led by Daniel. Hardware/PCB design, programming, and enclosures were split fairly evenly.

Conclusion - Daniel

The challenge to expand on the functionality of the given webcam was a valuable opportunity to fully integrate and prototype a multifaceted design. This required us to fully utilize our skills in areas such as PCB design, microcontroller programming, and in networking, which was a unique and rewarding experience. Though learning about a broad range of topics from server configuration to OpenCV was a difficult process, it was all extremely practical and resulted in a well performing webcam with distinct features. As an electrical engineer who is interested in computer science, I was very motivated to work on projects like designing a compact PCB and learning about Haar Cascades. I also enjoyed the integration process of this project, as it required a strong understanding of each component in the design.

Though we reached all of our goals, the time restraint of 10 weeks barred us from going even further. I would have liked to improve the overall stability of our stream either by using other components or improving our software. It would have been interesting to try more security features for the webcam, such as it unlocking a door when it recognized a face.

I would like to continue using OpenCV for a variety of other applications, such as face and object recognition, using tracking instead of detection at every frame, and being able to train the model ourselves. In addition, it would be a fun challenge to have the webcam mounted on a quadrotor or wheeled robot follow a person or object around.

References

[1] https://tools.ietf.org/html/rfc6455
[2] https://sublimerobots.com/2015/02/dancing-mustaches/

Full Code

MCU snippets have been included in the report. Full backend code can be found at https://github.com/fahadkh/kfcams.