Eyestrain-Alleviating Display Adjustment

By

Bryan Huang

Peter Siborutorop

Raghav Verma

Final Report for ECE 445, Senior Design, Spring 2021

TA: Bonhyun Ku

5 May 2021

Project no. 69

Abstract

Digital eye strain is a growing issue as increasing numbers of people spend an increasing amount of time looking at display devices. Our solution aims to address this by detecting the symptoms of eye strain, and adjusting display devices to relieve this strain. The ability to detect blink rates and ambient light levels as parameters help us determine the ideal display conditions to prevent eye fatigue.

1. Introduction

1.1 Problem and Solution Overview

Digital eye strain is a common condition that affects people who spend time using display devices. Digital eye strain has multiple negative effects on vision, and most people are potentially affected by it. The primary strategy for preventing digital eye strain is preemptive action — adjusting the viewing environment to minimize the potential for eye strain [1].

Two symptoms associated with digital eye strain are reduced blink rate and blink completeness. These symptoms are detectable through computer vision, and serve as a sign that the current viewing environment is not healthy for the user's eyes [1]. Our system uses a combination of computer vision and environment light detection in a system that will correct the user's viewing environment to accommodate any negative changes in eye strain.

1.2 Design Overview

Our design consists of three major blocks: The sensors, microcontroller, and software module.

The sensors, a camera and an ambient light sensor, read data on the user's environment and take video of the user's face. The data they collect is transferred to the microcontroller, which converts the data into a form readable by the software module being run on the user's device. This data is communicated serially via USB to UART and received by the software module. The software module then performs computer vision algorithms on the received video data to detect blinking. It uses that blink data, as well as the data on the user's environment, and uses them as parameters to determine the ideal display settings for relieving eye strain, then adjust the device displays accordingly.

1.3 Block Diagram

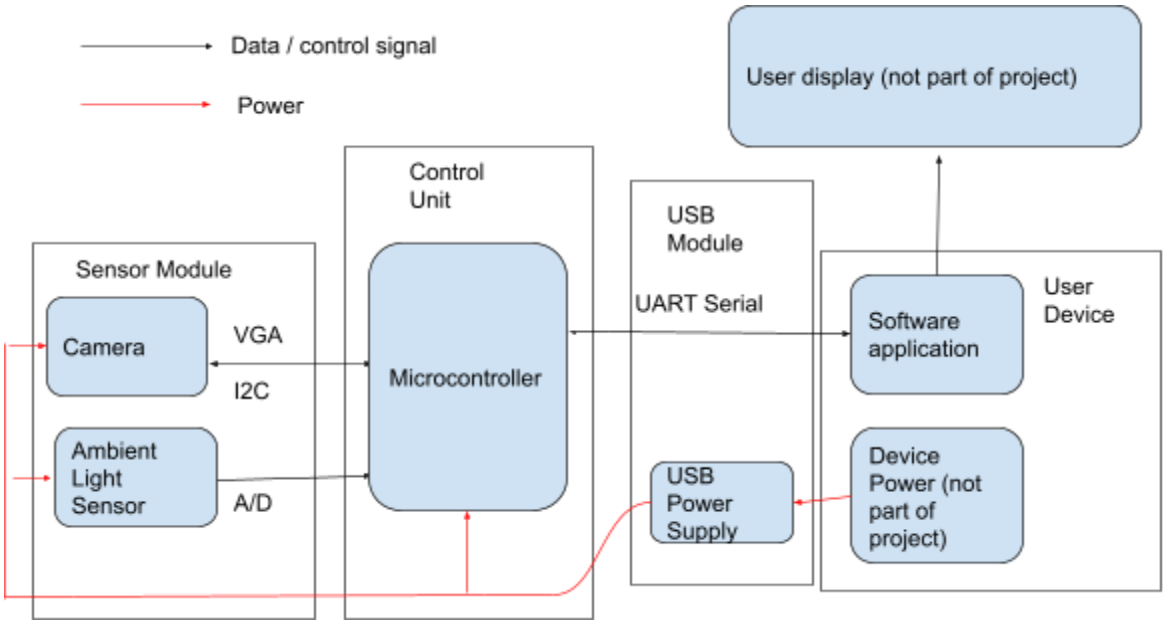

Figure 1. Design Block Diagram

1.4 Results

Our final product was capable of performing blink detection, reading input from the microcontroller's sensors, and adjusting the display accordingly. While all portions were functional, some issues in timing necessitated changes to the initial design. In addition, we discovered issues with our chosen hardware's ability to perform the tasks we wanted to assign to it. The details of these issues will be elaborated on in the rest of the report.

2. Design and Implementation

2.1 Design Procedure

There are a few changes made to our design since the initial proposal. In our original design, we planned to write the software in C++. We changed these decisions, with C++ being more difficult to prototype with, and having less compatibility with chosen computer vision tools, as well as our microcontroller.

Another change made was the operating system. Our original plan was to have the target device be a Windows device, but we used macOS in our final results. However, we found ways to develop equivalent software for both operating systems - it was mostly a matter of compatibility and convenience. The only compatibility issue lied in adjusting the displays themselves - all other software was fully portable between operating systems.

In our original design, all parts of the project were designed to run in real time at ten frames per second. We then realized that only the camera and microcontroller modules needed to run at 10 frames per second in real time in order to capture relevant blinking data. The logic behind this is that, because eye strain is slow to develop and does not need to be quickly addressed, there is no need to be on a constant lookout for symptoms. The software application was redesigned to only adjust the display every few minutes, so there was no need for it to continuously sample video. This avoided issues with skipping frames and relieved timing constraints on image processing.

2.2 Design Details

2.2.1 Camera

The camera we use is the OV7670 camera module. It provides VGA signals to the microcontroller, consisting of an 8-bit wide digital data signal as well as 3 timing signals, being a pixel clock, horizontal sync signal, and vertical sync signal. It also supplies a bidirectional pin for I2C protocol communication. In order to read a full image, our microcontroller must be able to read and store the data signal each time the pixel clock ticks, and signal the beginning of a new image whenever the vertical sync signal ticks.

The OV7670 is driven by an input clock signal XCLK. According to the data sheet [2], the minimum XCLK signal is 10MHz, but that is faster than the 8MHz our microcontroller can provide. The camera is still fully functional at 8MHz, however.

The camera contains multiple registers that can be written to/read from through the I2C protocol, allowing us to adjust multiple parameters, such as the color format for output, image resolution,

clock divider, as well as multiple preprocessing operations. We set the camera to provide output in the YUV color space. The YUV color space provides the brightness of the pixel in a 4-bit Y value. This allows us to easily take the grayscale image needed for blink detection while saving bandwidth and computation time compared to converting RGB pixels to a grayscale value. We intended to set the camera to 160x120 resolution output, but there were issues with timing that prevented full implementation. The details of this are explained in section 3.1.

2.2.2 Ambient Light Sensor

In our original design, we designed our system using the OPT3001 ambient light sensor. One big issue we discovered was that the OPT3001 required a clock input and the clock frequency didn't match the requirement that the OV7670 camera had.

f_{SCL}	SCL operating frequency	0.01	2.6	MHz

Figure 2. OPT3001 System Clock Frequency Range

f_{CLK}	Input Clock Frequency	10	24	48	MHz

Figure 3. OV7670 System Clock Frequency Range

Due to this, if we were to continue with the OPT3001, we had to operate both subsystems in different frequencies, where we would need two microcontrollers to operate both subsystems concurrently. The better solution we pursued was changing the ambient light sensor in use, where we changed the ambient light from the OPT3001 into the TEMT6000. The TEMT6000 doesn't require a clock input, thus this allowed us to operate the OV7670 with its required clock input, while obtaining the values for the ambient light sensor..

The TEMT6000 ambient light sensor detects the illumination at its surroundings, and converts it into a current measurement. The higher the illuminance in the room, the higher the value the ambient light outputs.

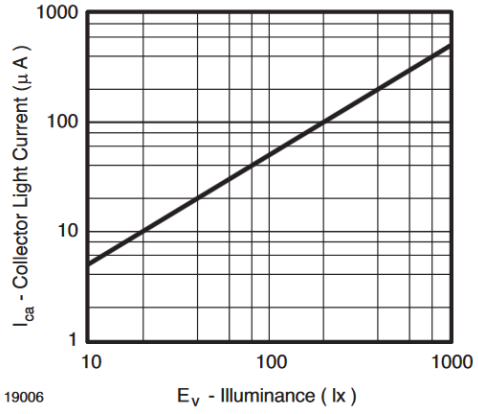

Figure 4. Ambient Light Sensor Measurement Graph

The ambient light sensor is connected to one of the analog pins of the microcontroller, where it is read through the analogRead() function. The value we obtain in the ambient light sensor through the microcontroller and sent to the PC.

2.2.3 Microcontroller and Microcontroller Software

The microcontroller in our original design was the ATMEGA32U4. The original reasoning behind the ATMEGA32U4 was to fit the original ambient light sensor and camera module input pins. Additionally, the ATMEGA32U4 had a built-in D+ D- converter, where we didn't need an additional TTL to USB module. The issue we had when implementing the microcontroller was that the microcontroller wasn't being read and programmed. The issue was that the D+ and D- weren't being powered correctly. The voltage reading on both connections was around 13mV, where the expected value should be around 3.1V. Due to this, we changed the microcontroller in design into the ATMEGA328p. This change was possible due to the change in the ambient light sensor, where the TEMT6000 required less input pins, thus allowing us to use ATMEGA328p instead of the ATMEGA32U4. The ATMEGA328p is able to send the data to the connected PC through a TTL to USB converter module. The microcontroller is then connected to the PC and programmed through the Arduino IDE.

The microcontroller is responsible for setting up the driving XCLK for the camera. By setting particular registers of the Arduino board, we can create an 8MHz clock output (half the clock of the microcontroller itself) to pin OC0A (digital pin 9) of the microcontroller. This clock output is produced by toggling the output each time the microcontroller's own clock cycle ticks.

Communicating with the camera is done in the setup phase of the software, using the I2C protocol to set the registers of the camera to our previously described settings. We simply use the Arduino's Wire library to set these registers to fixed values. The amount of time this communication takes is not a limiting factor on operation, so we ignore any potential optimization in this portion.

Reading the data is done simply by performing reads on the input pins. For reading the image data from the OV7670, using the Arduino's provided digitalRead() functions is too slow. Because we need multiple digital pins at once, we instead copy from the port input registers, allowing us to read pins that share a port simultaneously. Our final design has us read from 3 bits on port C, 4 from port D, and 1 from port B. Some of these bits include the color signal of the YUV image, which we can freely ignore, but we chose to read them in for debugging purposes, in case of the need to read color images.

Writing the image data to the target device is done simply by signalling the start of an image with an initial string to the Serial output, and then writing the raw image as bytes to the Serial. The same is done for the ambient light data.

2.2.4 Target Device Software

The software implementation had four major components whose function was threefold: to enable the user to interact with the system, to detect the blink rate of the user from raw camera input, and to change the display's brightness accordingly. This section will elaborate the individual components that enabled this function.

2.2.4.1 Hardware Software Integration Module

In order to communicate with the microcontroller, the software has to be able to read serial usb messages at the baud rate the microcontroller is outputting. In order to achieve this, we used the module pyserial to query the inbuilt ports in the host computer and read data if provided.

```
#Modify to get row by row instead of whole thing at once
@break_after(20)
def serialReadImage():
        s = Serial(port=getPorts()[-1], baudrate=1000000)
        while True:
            val = str(s.readline())
            if val.find("image") != 0:
                break
        val = s.read(320*240)
        if val:
            intVals = [x for x in val]
            print(intVals)
            arr = np.array(intVals).reshape(240, 320)
            arr = arr.astype(np.uint8)
            im = Image.fromarray(arr)
            im.save("imageRead.png")
            return
```

Figure 5. Example code from Hardware Software Integration Module to Read Image Being Sent Serially and Save to Disk

2.2.4.2 Eye Strain Detector

After reading an image in, the next step is to detect the number of blinks the user has over a given time period. For this step in the process, the blink detection module is used. The following is a description of the blink detection algorithm as implemented in code.

The first step of the design is to capture and load relevant video. Currently, I am using my webcam to record this video and immediately saving it to disk in 20 second increments. I determined that the minimum time necessary to accurately calculate blink rate is 15 seconds to compensate for fluctuations that may occur if a smaller sample is taken. To be safe, I am currently using five seconds above that time. After recording, the video is loaded onto the next portion of the pipeline and analyzed frame by frame for its full length.

The first step when receiving a frame is to convert the image to grayscale. This is done for many reasons. Analysis of edges is easier in grayscale since there is only one color channel, and facial and eye detection without neural networks uses edge detection to perform a significant portion of its determination. In addition, grayscale images require less time to analyze since they are inherently smaller than their colored version. In addition to converting the image to grayscale, I increase the exposure of the image since I have noticed it improves the edge detection process as well.

Once the image has been preprocessed, the next step is to begin detecting. The algorithm detects in three steps: the first step is to detect a face in the image, and if present detect the eyes, and then lastly detect if the eye is open or closed. To detect the face, I used a pre-trained haar cascade from opencv [7] to create the boundary box for the face. I then make this boundary box the new bounds for the image, and thus discard any data outside the box. Note that if there is more than one face present, I choose the largest face (or the boundary box with the most area) since this algorithm is only designed for the use case where there is one user. If, for whatever reason, a face is not detected I move on to the next frame. I keep track of the percentage of frames in which a face is detected for testing and improving the face detector.

Once we have the face image, the next step is to find the eyes. For this part, I use dlib's haar cascade [8] that is pre-trained on eye data. I chose this model because I compared this with opencv's and it is more accurate. Similarly to the face detection portion, this model also returns boundary boxes for the eyes. If, for whatever reason, a single eye is not detected I move on to the next frame and record this failure for future analysis. To determine if the person is blinking, only one eye needs to be present. However, if both eyes are successfully detected I will use both of the eyes to get a more robust detection. Once again, I discard the face data and only keep the eye data for the next portion of the algorithm.

The final part of the detector is to determine if the given eye data is open or closed. To do this, I perform a color histogram analysis [9] of the eye image. If a certain threshold of white is met (which represents the white in the eye), I deem the eye open. Otherwise, the eye is closed. If the eye is deemed closed, I put in a 300 millisecond wait until the next closed eye can be detected to prevent multiple reads from the same blink.

For the whole video, I keep track of the number of closed eyes = number of blinks found. After every frame of the video is analyzed, the blink rate is calculated using the following equation:

Blink Rate (blinks/minute) = (Number of Blinks Detected) * 1 / (Duration of the Video in Minutes).

For the 20 second video I used, the blink rate is essentially three times the number of blinks detected. The last step is to convert the blink rate in blinks/minute into eye strain. To do this, I just set a few thresholds that map to ranges within blink rate as such:

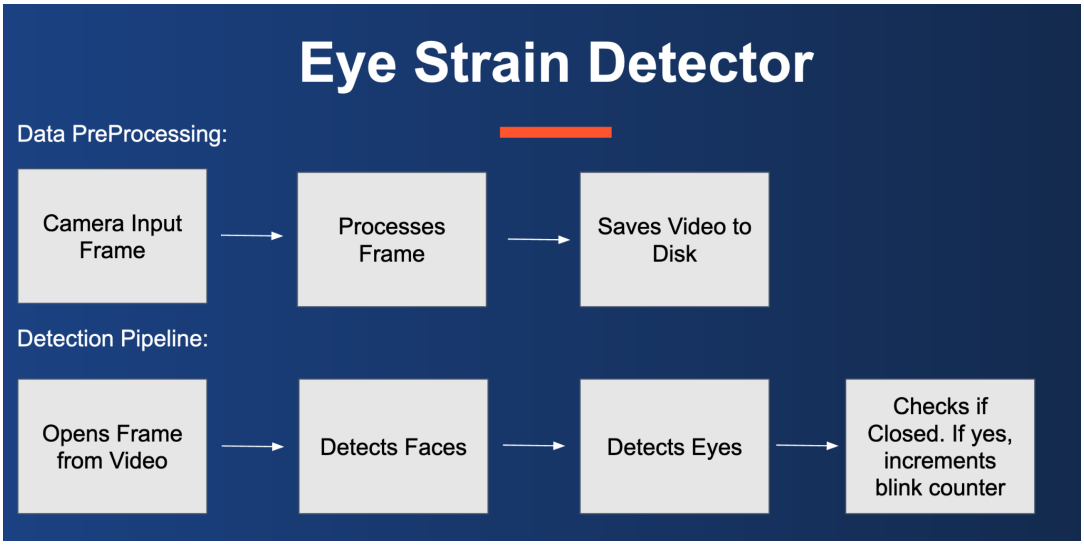

Figure 6. Core steps in the blink detection pipeline

Figure 7. Sample frame where a blink is detected

2.2.4.3 Brightness Adjuster

Once a level of eye strain is detected, the next step in the process is to adjust the display's brightness accordingly. The first step in order to do this is to convert the user's blink rate to a level of eye strain. This is done through simply mapping certain ranges in blink rate to a corresponding level of eye strain. The formula used to do this is shown below.

```
def convertBlinkRateToStrain(blinksPerMinute):

    if blinksPerMinute < 8:
        return strains["high"]
    elif blinksPerMinute < 10:
        return strains["medium"]
    elif blinksPerMinute < 13:
        return strains["low"]
    else:
        return strains["none"]
```

Figure 8. Conversion of Blink Rate to Eye Strain

Next, the display's brightness must be reduced by the appropriate amount. This is also done by another map from level of eye strain to change in display brightness. If the user's eye strain is high the brightness is lowered by 20%, if it is medium the brightness is lowered by 15%, and if it is low the brightness is lowered by 10%. If no eye strain is detected the brightness is not changed. In addition, the brightness module takes into account changes in ambient light. When the ambient light changes significantly, the display adjusts accordingly (e.g. ambient light decreases by a given amount so the display's brightness decreases by a given amount).

2.2.4.3 User Interface

The last required portion for the software component of this project is the GUI. The user needs to be able to interact with the software to use the features they provide. To do so, a GUI has been developed as such:

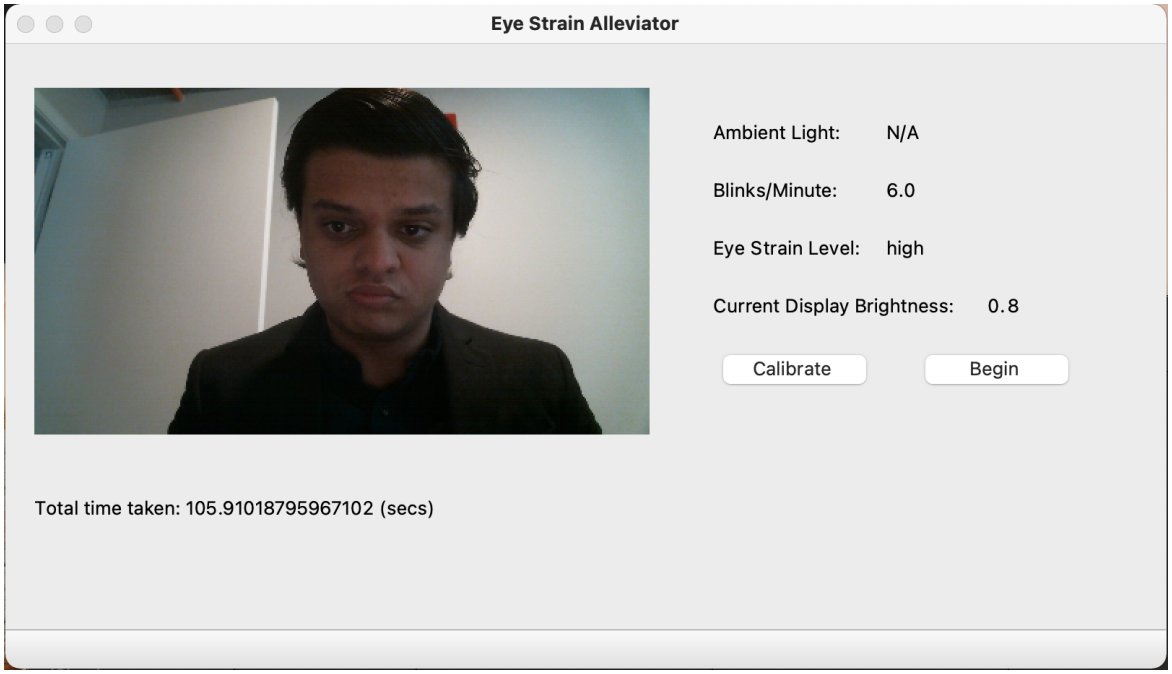

Figure 9. Sample Image of the GUI after Running the Blink Detector (Pressing Begin Button)

The core features of the GUI include a live feed of the camera that will be used to run the blink detector, labels that pertain to various stats about the ambient light and blinking, and two buttons. The first button is the calibrate button, which reads the ambient light data from the ambient light sensor and decides if the monitor's brightness needs to be adjusted. The second button is the begin button, which runs the blink detection pipeline: record 20 second video, detect number of blinks in that video to get blinks/minute, and change monitor's brightness according to blinks/minute. This process usually takes a little over 100 seconds, as the figure above shows.

3. Verification

Because our microcontroller and ambient light sensors worked entirely as intended, verification details are in Appendix A.

3.1 Camera

Our camera module was the only module to fail its verifications entirely. Its intended frame rate was 10 frames per second. While theoretically possible as shown by other existing projects [3], there were multiple factors that prevented us from accomplishing our target frame rate. The most important is the nature of the OV7670's output. The OV7670 outputs VGA-like control signals, and the timing of these signals is driven by the camera's internal clock. We can adjust the internal clock rate by setting a register holding the division rate from the camera's driving XCLK

to the output PCLK. In testing, we found that, for our 8MHz driving XCLK, we could theoretically achieve ~10 low-resolution frames per second by setting the divider to 4, or a 2MHz PCLK. Reading the PCLK correctly also involves being able to read whether PCLK is in the high or low portion of its clock cycle, so essentially our microcontroller needed to read these control signals at 4MHz. The microcontroller's clock rate is 16MHz [4]. Most C statements translate to multiple instructions, and therefore multiple clock cycles, often more than 4 per instruction. This means we cannot check the control signals and make decisions within a single PCLK cycle. While we can theoretically use assembly NOP commands to wait for a known number of clock cycles, debugging this proved too complex and inconsistent for us to accomplish, especially with wiring issues when not working on the finished board.

We could verify the camera's ability to capture images in general, but the resulting frame rate was very slow (<1 frame per second).

Figure 10. Camera Image Result

3.2 Target Device Software

For verification, the software was vital in allowing us to confirm the efficacy of the other modules, using its console output to read test results.

We also tested the software itself, in particular, we tested the accuracy and execution time of its blink detection. Using a dataset of images with known eye blink status (closed eyes, open eyes, no eyes), we tested the average accuracy and execution time for the images. We tested the data set multiple times on different parameters to verify the efficacy of our low-resolution camera. The camera's maximum resolution of 640x480 was downscaled using nearest-neighbor to simulate a lower camera resolution. The histogram-of-oriented-gradients face detector is sensitive to the scale of the image, and needs to operate on a minimum face size [5]. By

upscaling the image from the low camera resolution, we can make the blink detector more accurate, without having to load larger images.

Camera Resolution	Upscaled Resolution	Average Execution Time / Image	Accuracy
640x480	960x720	106.5 ms	0.84375
320x240	960x720	106.2 ms	0.8125
160x120	960x720	106.1 ms	0.8125
640x480	640x480	51.8ms	0.78125
320x240	640x480	51.6ms	0.78125
160x120	640x480	51.8ms	0.8125

We found that accuracy and execution time were more dependent on the resolution we upscale the captured image to than the camera resolution. Not only that, but the accuracy was generally within acceptable parameters for both resulting upscaled resolutions. While the blink detection itself no longer needs to have fast execution time in our final design, 640x480 resolution is fast enough to run in real time if needed.

4. Costs

4.1 Labor Costs

Our labor costs were a little lower than expected. Each of us put in 100 hours of work to complete this project. Thus, our total labor cost is estimated as follows:

3 people * 100 hours each * $44.18/hour * 2.5 = $33,134.

Note that this hourly rate is calculated using data on starting salaries for computer engineering and electrical engineering as calculated in our design document.

4.2 Component Costs

Our component costs were as such:

Part	Cost	Quantity	Total Cost
Microcontroller (Arduino ATMEGA328P-PU)	$2.52	5	$12.6
PCB (Custom Design)	$1	15	$15
Camera Sensor (OV7670 - subject to change, available for checkout from ECE445 inventory)	$4.50	2	$9.00
Ambient Light Sensor (TEMT6000)	$1.39	2	$2.78
Resistors, Capacitors, Crystal, LED	$8	1	$8
TTL to USB Module	$4.94	1	$4.94
Total			$49.32

4.3 Total Cost

Thus our total cost is $33,134 + $52.32 = $33,183.32.

5. Conclusions

In conclusion, our project functioned properly and met all the high level requirements necessary to consider it functional. The only component of our project that did not fully function was the OV7670 camera sensor. Unfortunately, we could not get this sensor to output data at the target frame rate and thus were not able to integrate it with our final product. Instead, we opted to use an external webcam for the final product so that all other portions of the project could function.

For future improvements, we would like to select a better camera sensor and microcontroller that are more suited for our verification requirements, create an physical encapsulation of the product so that it can be commercialized, and improve the blink detection algorithm so that it can run faster (possibly real time) and be used for other proposes (blink detection is an open area of study used in many applications beyond eye strain detection).

We had a blast working on this project and it taught us a lot about how to create a product from conception to product demo. Along the way, we learned how to work effectively as a group and because of that were able to successfully create our product. Thank you especially to Bonhyun, all the other TA's, and the professor for helping us along the process and making this a worthwhile experience.

References

[1] Coles-Brennan in Clinical and Experimental Optometry, ‘Management of digital eye strain’, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1111/cxo.12798. [Accessed: 17-Feb-2021]

[2] OmniVision, ‘OV7670/OV7171 CMOS VGA (640x480) CameraChip™ Sensor with OmniPixel ® Technology’, 2006. [Online] Available: http://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf. [Accessed 20-Apr-2021]

[3] indrekluuk, ‘LiveOV7670’, 2020. [Online] Available: https://github.com/indrekluuk/LiveOV7670 [Accessed 20-Apr-2021]

[4] Arduino, ‘Arduino Uno Rev3’, 2021. [Online] Available: https://store.arduino.cc/usa/arduino-uno-rev3 [Accessed 4-May-2021]

[5] Dlib C++ Library Example Programs, ‘face_detector.py’, 2021. [Online] Available: http://dlib.net/face_detector.py.html [Accessed 4-May-2021]

[7] Mordvintsev, A., & Revision, A. (2013). Face Detection using Haar Cascades — OpenCV-Python Tutorials 1 documentation. OpenCV. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html

[8] Chaudhari, N. (2020, July 14). Blink Detection using Python. | AlgoAsylum. Medium. https://medium.com/algoasylum/blink-detection-using-python-737a88893825

[9] D, R. (2019, January 29). Image Histograms in OpenCV - Raghunath D. Medium. https://medium.com/@rndayala/image-histograms-in-opencv-40ee5969a3b7

Appendix A: Requirement and Verification Tables

Requirement	Verification	Verified?
1. The camera should be able to capture images at a rate of at least 10 frames per second.	1. Have the camera continuously capture images for a set time, and calculate the number of images taken per second.	No

Requirement	Verification	Verified?
1. The ambient light sensor delivers a signal with levels that can be read in software on the microcontroller.	1. Expose the sensor to different brightnesses and show the change in its output signal.	Yes

Requirement	Verification	Verified?
1. The microcontroller can deliver sensor inputs to the PC via USB.	1. Have the microcontroller capture known inputs and show they are being sent through serial output.	Yes

Requirement	Verification	Verified?
1. The blink detector must be able to correctly detect blinks at least 75% of the time. 2. The blink detector must be able to run detection on the images at a rate of 10 frames per second.	1. Run the detector on a known dataset. At least 75% of the images should be correctly categorized. 2. Use a timestamp to record the time it takes to run each image at the target resolution.	Yes Yes

Requirement	Verification	Verified?
1. The brightness of the target device should be changed based on the level of eye strain and ambient light sensor.	1. Feed multiple blink rates and light levels to the system and show it affects the brightness.	Yes