[This document describes the final written report required for your Senior Design(ECE/COE 1896) project. Use this template as a format for your report. Delete/Remove sections that do not apply to your project. Do not change the formatting in the template.]

[Editorial comments appear in this style (italic blue font in brackets, hidden font) and will not normally print with your document. You may delete the editorial comments after you understand them.]

ECE/COE 1896

Senior Design

Eye Tracking Headset for Strabismus and Amblyopia

Prepared By: James Liu

John Marshall

Sean McCarthy

[This page is intentionally blank.]

# Table of Contents

*[When you want to update your table of contents (i.e. prior to turn-in), Right-click on table, select ‘update field’ and select ‘update entire table’]*

[Table of Contents i](#_Toc500785471)

[Table of Figures ii](#_Toc500785472)

[Table of Tables ii](#_Toc500785473)

[1. Executive Summary 1](#_Toc500785474)

[2. Problem Definition 1](#_Toc500785475)

[3. Background 2](#_Toc500785476)

[4. System Requirements 3](#_Toc500785477)

[4.1 Hardware 3](#_Toc500785478)

[4.2 Software 3](#_Toc500785479)

[5. Design Constraints 4](#_Toc500785480)

[5.1 Hardware Constraints 4](#_Toc500785481)

[5.2 Software Constraints 5](#_Toc500785482)

[6. Technical Standards 6](#_Toc500785483)

[6.1 Standards Selected 6](#_Toc500785484)

[6.2 Scope of Standards 6](#_Toc500785485)

[6.3 Brief Description 6](#_Toc500785486)

[7. Evaluation of Design Concepts 7](#_Toc500785487)

[7.1 Hardware 7](#_Toc500785488)

[7.2 Software 8](#_Toc500785489)

[8. Team 9](#_Toc500785490)

[8.1 James Liu 9](#_Toc500785491)

[8.2 John Marshall 9](#_Toc500785492)

[8.3 Sean McCarthy 9](#_Toc500785493)

[9. Final Prototype 10](#_Toc500785494)

[9.1 Hardware 10](#_Toc500785495)

[9.1.1 PCB Board 10](#_Toc500785496)

[9.1.2 3D Printed Headset 12](#_Toc500785497)

[9.2 Software 12](#_Toc500785498)

[9.2.1 Image Processing Algorithm 13](#_Toc500785499)

[9.2.2 Pupil Detection and Analytic Algorithm 14](#_Toc500785500)

[10. Testing, Data Analysis and Results 17](#_Toc500785501)

[10.1 Test Plan 17](#_Toc500785502)

[10.2 Hardware Testing 17](#_Toc500785503)

[10.3 Software Testing 18](#_Toc500785504)

[10.3.1 Pupil Detection Testing 18](#_Toc500785505)

[10.3.2 Baseline and Fake Desynchronization Testing 19](#_Toc500785506)

[10.3.3 False Positive Testing 20](#_Toc500785507)

[11. Project Timeline 22](#_Toc500785508)

[12. Conclusions and Future Work 22](#_Toc500785509)

[13. References 23](#_Toc500785510)

[To update the Table of Contents, Table of Figures, or Table of Tables, click on any text within the table. Press the F9 key on your keyboard (or right click and select “Update Field”). Then select “Update entire table” from the pop-up window and click “OK”. If you use the formatting styles as directed below, your tables will fill automatically and have the correct page numbers.]

# Table of Figures

**Figure 1**: An early version of what became the current pupil detection algorithm. The image was taken with a visible-range camera. The blue dots represent “hits” in the raster scan and the red dot represents the mean value. 7

**Figure 2**: Solidworks Model of the Glasses Frame 8

**Figure 3**: An early version of what became the current pupil detection algorithm. The image was taken with a visible-range camera. The blue dots represent “hits” in the raster scan and the red dot represents the mean value. 10

**Figure 4**: Realization of the PCB Board 11

**Figure 5**: Realization of the PCB Board 12

**Figure 6**: Flow Chart Summary for Software Design 12

**Figure 7**: The remapped values from trans, showing how the range of pixel values 0 - 255 in A become -1 - ~-.14 14

**Figure 8**: The input, A, left, and the output, trans(A), middle. Notice how the shading details on the skin and sclera are greatly diminished, the iris becomes much lighter, and the pupil remains dark. The final new\_pupil result is on the right. 15

**Figure 9**: Two frames taken at the Design Expo with missing data, one with a false blink and the other with a correctly identified pupil 17

**Figure 10**: 1: report for ts\_1, the simulated lazy eye. 2: report for ts\_2, the case of no movement. 3: report for ts\_3, the simulated nystagmus-like twitch of the left eye. 18

# Table of Tables

**Table 1**: Data Collection Cohort at Senior Design EXPO 19

[If necessary, insert a blank page after your Table of Tables to make the front matter end with an even page. You should be able to print your final report double-sided and have the page numbers appear as they would in a book.]

[The following sections must be included in your final report. You may add sections or subsections to suit the needs of your project, but do not omit any of the sections provided here, unless instructed to do so. Follow the instructions and examples in each section to build your report. If you insert material from other sources, make sure the format is consistent with this document’s original format. If you have trouble with the formatting, please consult your course instructor.]

# Executive Summary

Eye diseases like Strabismus and Amblyopia are very common in both third and first world countries. Despite advances in medical technology, detection for eye conditions is both difficult and time consuming. Our solution to this problem is the fabrication of an advanced eye tracking headset and accompanying pupil detection and analytical algorithm software to eliminate the need for routine visits to the physician’s office. Alternatives such as Pupil from GitHub as well as other software and hardware packages were considered. Investigation revealed that none of these alternatives have the same versatility, ease of use or specific application that we desired.

Design encompassed several parts spanning both hardware and software. A physical frame needed to be custom made using a 3D printer and Solidworks to allow for the integration of multiple components into an ergonomic device. A custom PCB board was completely soldered and incorporated. A pupil detection program was written from scratch in the Python programming language. An analytical algorithm was also written from scratch in Python. All of these components were integrated and made compatible with each other. Constraints that limited the performance of the device include the limited resolution of the camera as well as the limited processing power of our personal computers. A non-technical constraint, the decision behind turning this eye tracker into a monitoring, but not diagnostic, tool, also emerged. This will be further elaborated upon in the Design Constraints.

This device was tested with data collected from a cohort of 20 subjects. False positive testing was the bulk of our experimentation. In addition, artificial data resembling a patient with strabismus and amblyopia was simulated. Resulting conclusions found that the device performs very well, being able to accurately find areas and frames of desynchronization and being able to recognize healthy individuals. These results exceeded our original objective and opened up possibilities for further design to expand the device. Future work will be focused on reducing the processing time and expanding the application to other horizons.

# Problem Definition

Our project involves creating a pair of glasses that track eye movement to monitor irregular ocular conditions like amblyopia (crossed eyes) or strabismus (lazy eye) that cause desynchronization between the two eyes. This device and accompanying software will be able to provide a user with information about irregular ocular behavior, which can help people with specific conditions monitor their wellbeing.

We hoped to be able to complete both a working pair of infrared eye tracking glasses and a software package that receives and analyzes ocular behavior from the glasses. The specific goals sequentially were: outfit a pair of small webcams with infrared filters, read in the video from the camera to a computer, determine whether there is amblyopia or strabismus occurring, and analyze whether this is occurring with greater frequency than normal for the user, and finally write a basic UI that outputs this information to the user. Thus, the specific hardware designed is the 3D printed glasses as well as the PCB board. The software designed encompasses both a pupil detection program and the analytical algorithm, as well as a program to complete the JPG images.

It is important that our device will be used to monitor an existing or suspected condition, not to diagnose. This means that our design will not involve attempting to fit certain patterns of eye behavior to any pre-existing diagnostic data set; instead, it will alert the user of the presence of symptoms. Tracked over many sessions, this can provide a picture of whether conditions are improving or worsening.

# Background

Eye tracking is becoming an increasingly popular consumer and medical technology and is finding new applications constantly. One novel application explored by the University of Melbourne in conjunction with the company Tobii was the use of eye tracking for diagnostics. Many diseases and conditions, from inner ear inflammation to lazy eye to dystonia, can cause irregular behavior in the eye. Tobii is currently working on using eye tracking headsets to diagnose nystagmus. However, there are no widely available such headsets that are used instead for regular monitoring. Monitoring of nystagmus, for example, is useful because it can signal other long-term issues like cataracts or astigmatism. In a developing child, monitoring of eye movement can be used for early detection of improper ocular development.

Our design will be unique because it will couple a pair of tracking glasses with a monitoring software package that will analyze movements to determine whether patterns of irregular behavior like nystagmus or strabismus are occurring. For people who already have a known ocular condition, observing that these symptoms are occurring more or less frequently can give insight about the efficacy of medication, the worsening or improvement of a condition, etc.

# System Requirements

## Hardware

A system requirement is that the Arduino IDE is needed to control the microcontrollers. The ATmega328P microcontroller is compatible with Arduino IDE. Arduino IDE is easy to use and write. The ATmega328P is was found to be a good microcontroller with accessible pins, which made it easy to wire the camera and the other components to the ATmega328P chip. The button is hooked up to pin 9. This controls the state of the microcontrollers. A red LED id was connected to pin 10 to show the user when the camera is ready to take pictures. Jumper pins are used to connect pin 2, pin 3, 5 volts, and ground to the Camera. The Rx line of the Camera must be wired to the Tx line of the microcontroller. The Tx line of the camera needs to be connected Rx line of the microcontroller. The six pin male jumper connects the microcontroller to the serial port of the Raspberry Pi using a FTDI cable.

## Software

The user does not need a Raspberry Pi to run the software program. Any computer or device with Python 2.7 or later and two USB ports, needed to read in the serial data, can run it. The requirement of two USB ports instead of one is due to the nature of the microcontrollers of the PCB board being set to run in parallel, which allows a frame from each eye to be captured at the same time. Because the primary limiting factor in the speed of the software is the baud rate of the cameras, the image processing and pupil detection algorithms only being run after the completion of the serial read, the choice of computer will not significantly increase or decrease the overall speed of the program. However, the choice of operating system (Windows, Linux, iOS) is important because each has a distinct procedure for naming and mounting serial ports. The code was written and compiled on a Debian operating system. Should a user run the program on a different OS (i.e. non-Unix-based), he or she will need to rename the serial port paths in the beginning of the capture code.

The Python libraries and packages required by the program, NumPy, SciPy, binascii, PIL, MatPlotLib, and serial, are already included in most Python builds. Use of more complicated packages like OpenCV was avoided because of the requirement of installation, the computational slowness of many in-built functions, and the usage of RAM on the Pi being too costly. In earlier versions of the pupil detection algorithm, a circle Hough transform from the OpenCV library was used, but this was abandoned because it ran at a prohibitively slow speed on the Raspberry Pi. The algorithm that was used instead is explained in the “Final Prototype” section.

Another small requirement of the code is the presence in the installation directory of three text files: “sread2.txt”, “sreadr2.txt”, and “history.txt”. The first two store the most recent serial port outputs while the last stores the desynchronization magnitude history for all previous runs; these are read and plotted by the analysis program. If they are not already present, the user will need to create blank text files with these names.

Once the Arduino code on the microcontrollers has been set to run for the desired number of frames and the serial port paths are correctly identified, the user need only run a Python script that reads from the serial ports, converts the images, runs pupil detection, and finally returns a readable printout in the Python shell of the analytic results. In the current version, a script that serves this purpose is expo\_demo.py, which was the demonstration code run at the Design Expo and on demo day.

# Design Constraints

## Hardware Constraints

One major constraint of the project is the camera. The rate of data flow from the camera impacts the overall time of project run time. The baud rate is fixed at 38400 baud from the manufacturer’s default setting. The camera sends the image, encoded in a hexadecimal format, to the microcontroller. Since the baud rate is fixed, the data cannot be sent any faster. The baud rate between the microcontroller and the computer can be changed but the data from the camera is fixed, so the data can only be sent as quickly as it comes in from the camera. For one image it takes about 20 to 25 seconds for the entire hex string to be sent to the microcontroller then to the Raspberry Pi.

A constraint that relates to the camera is the fact that the hex string still needs to be converted back into an image. Each image roughly has 25,000 thousand to 30,000 thousand hex characters. Sometimes, the camera would skip data, leaving portions of a frame unfilled and unconvertible. In this case, the length of each string of hex was slightly different. However, it was seen that for the first frame, there was reliably nothing wrong because each string started with “FFD8” and ended with “FFD9”. This the proper hex code to start and end an image.

Another constraint is that two cameras are needed to take an image of each eyes at the same time. In other words, a microcontroller is needed to control each camera. If one microcontroller was used in the project one camera will capture its image send the data then the other camera can take the image. This would slow the overall running time of the project. An image cannot be saved because there is no dynamic storage in the camera, so the data of the image needs to be sent as soon as the image is captured. There is some dynamic storage in the Arduino but it is roughly one kilobyte. Each image is roughly 15 kilobytes, so the image cannot be saved in the microcontroller. Therefore, two microcontrollers were used to send the data through two serial ports in parallel.

## Software Constraints

Early in the software design, the use of a Raspberry Pi placed unexpected constraints because of RAM limitations. As mentioned in the previous section, the original intent was to use functions from the OpenCV library. While trying to perform a Unix installation of the library, the Raspberry Pi ran out of memory. When a pared down version was finally installed, the circle Hough transform function ran extremely slowly (nearly five minutes per frame). When a newer design was attempted using a Sobel-Feldman transform (similar to a gradient operation, used for edge detection) from the SciPy library, the result was significantly faster but ultimately very slow. As a result, the decision was made to use only more optimized array operations from the NumPy library to build the pupil detection algorithm, namely the mean, median, arctan, and exp functions, as will be discussed later.

Another unexpected constraint emerged while trying to build the image processing algorithm and run the code with the IR cameras. Because the cameras would often leave frames incomplete because of a timeout, or they would complete an image with an insufficient number of scanlines (i.e. with missing data), the images would not be viewable on any conventional image viewers, such as MatPlotLib’s PyPlot library or the default Image Viewer program on the Debian OS. A setting from Python’s PIL library, LOAD\_TRUNCATED\_IMAGES, is set to TRUE in the include statements in the code, allowing the images to be processed as arrays by NumPy, but they are not available as complete, manipulable JPGs for use outside of the algorithm (unless a similar program for reading truncated images is used). However, there is an available script called pupilfromserial.py that allows the user to save complete JPGs of the frames from the most recent run with the pupils marked. These will be sent directly to the installation directory unless the path is changed in the code and can be read by any conventional image viewer. They are not necessary for the analytic program, but can be useful for demonstration.

An ethical constraint placed on the design concerned its functional purpose. Although it was originally planned as a diagnostic tool for conditions like strabismus, nystagmus, and amblyopia, it became clear that the device would need a more sophisticated algorithm designed with substantial medical data to create a product that could be said to “diagnose”. Instead, the focus became creating a design for a monitoring tool; the purpose of the program was not to analyze the behavior of the desynchronization but instead to assign it relative values so that, if the headset were worn regularly, it could be used to monitor whether such desynchronization was getting better or worse. An example of an application for this sort of device would be for individuals with concussions or risk factors for optic nerve atrophy or another relevant neurodegenerative disease; in these cases, a worsening of strabismus or amblyopia can be a sign of complications requiring medical intervention.

# Technical Standards

## Standards Selected

The three standards that we decided to include are the Vision Council standards and dimensions for Quality eyewear [1], the International Council of Ophthalmology standards for ethical eye treatment [2], and finally a specialized standard that the IRB uses to govern human data collection [3].

## Scope of Standards

These standards encompass the three fields of science that are most applicable to our device. Unfortunately, since Eye Tracking is at the forefront of engineering and medical technology it does not yet have specific standards of practice that apply it it. However, the scope of the selected standards encompasses proper eyewear, ethical ophthalmology practice and the IRB’s standards for human data collection.

## Brief Description

The first standard is the dimensions and proportions that are mandated by the Vision Council. Specifically, it outlines the length, width and height of glasses that are acceptable to be prescribed. This applies to our 3D Eye Tracker because we plan on merging the infrared cameras onto a custom frame. This frame will be designed using Solidworks to allow the compatibility of multiple components into an ergonomic and efficient singular device.

Second, the International Council of Ophthalmology states that physicians must inform patients of any risks with a certain procedure as well as using proper materials in their treatment. That is, materials that are not overly damaging or irritating to the eyes. This is in accordance with the performance and testing of our device. The cameras do not come into contact with the patient’s eyes. The headset was made to be as comfortable as possible. All lights on the infrared cameras were switched off so they don’t induce any headaches.

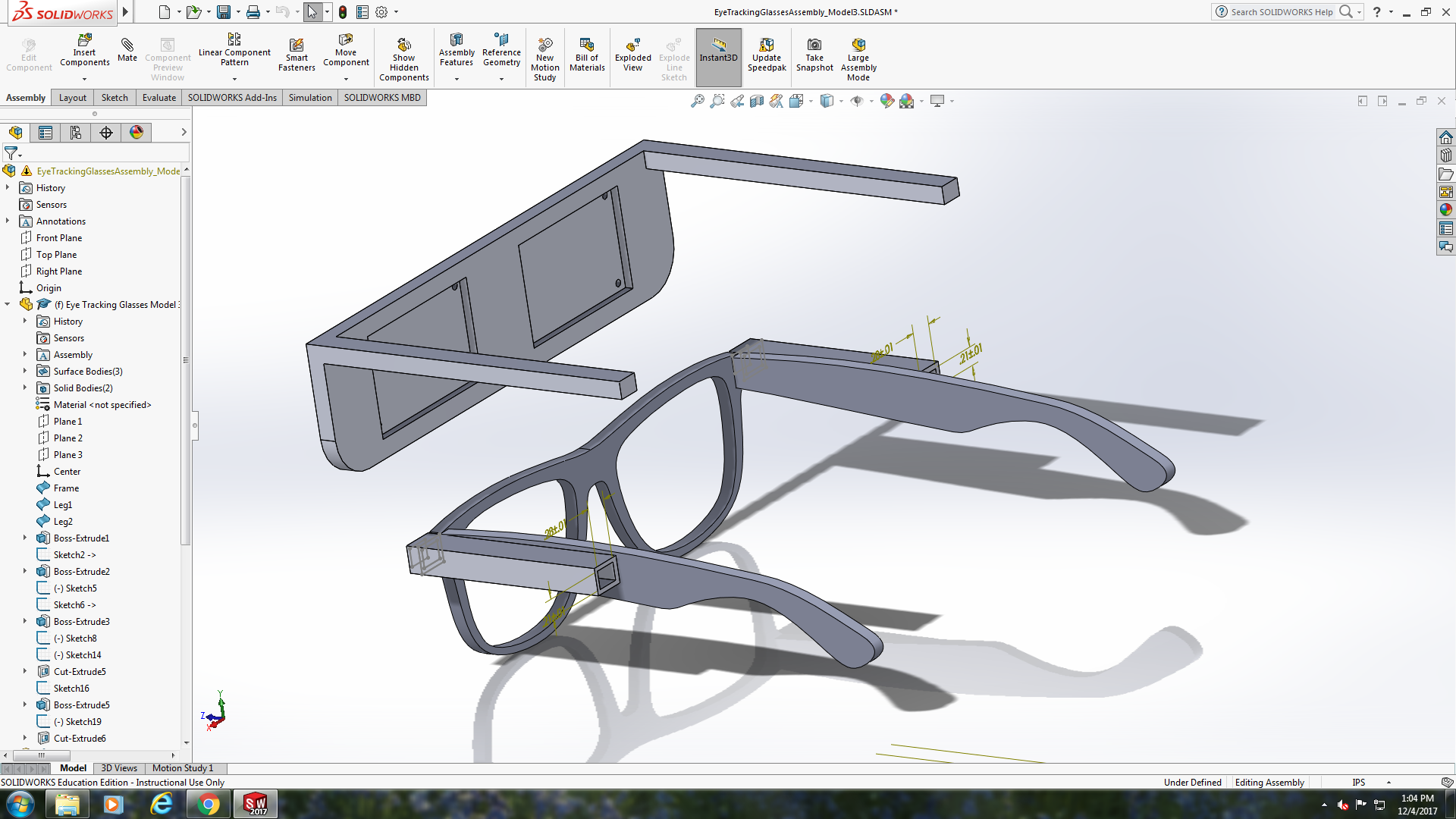
Finally, there are the standards of collecting data from human test subjects, which is regulated by the IRB standards of research. In summary, these state that every subject in a human cohort must be advised of the risks and benefits of their participation. They also must sign an explicit form of permission. During the study, they must not be subjected to unnecessary stress or harm. Dr. Akcakaya encouraged us to strongly consider this as our final testing involved data collection from a group of healthy test subjects. This was to test for false positive. Also, the data sets for strabismus and Amblyopia were artificially simulated by moving the frames, as is discussed more thoroughly in the Testing section. On behalf of Professor Dickerson, these standards did not apply to us as we were conducting this project for a university-sponsored course rather than for a research project with patent rights.

# Evaluation of Design Concepts

## Hardware

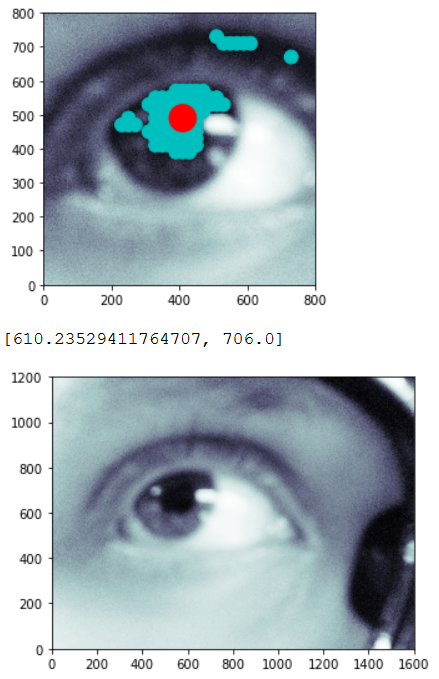
The first plan was to use the Raspberry Pi board with Raspberry Pi cameras that could be read straight into Python with the PiCamera library. However, a custom circuit was needed to fulfil the design requirements of the class. Also, a circuit that had more custom design to the camera circuitry was necessary. Designing a camera itself was not a part of the project but using a module and a microcontroller allowed for more custom design in the hardware. The plan remained that the camera module would still be wired to the microcontroller, which in turn would feed this data to the serial ports of the Raspberry Pi.

The original headset was planned to utilize a pair of store-bought glasses with camera attachments. However, later designs incorporated a 3D printed frame to allow for fine adjustment of the camera mount as well as a more user-friendly interface. Below is the model for the eye tracker.



**Figure 1**: Solidworks Model of the Glasses Frame

## Software



**Figure 2**: An early version of what became the current pupil detection algorithm. The image was taken with a visible-range camera. The blue dots represent “hits” in the raster scan and the red dot represents the mean value.

The earliest version of the pupil detection algorithm used a circle Hough transform from the OpenCV library. The results that this algorithm yielded for the frames captured at that time, from the Pi cameras, were very inaccurate, but more importantly, the algorithm ran extremely slowly. To process a single frame with a resolution of 1600x1200 took around seven minutes. Therefore, this program was dropped. The decision was made not to use the OpenCV library at all, in part because it was too taxing on the Raspberry Pi’s CPU.

The next version of the pupil detection algorithm, like the one shown in the figure above, used a Sobel transform to find a bounding box in which to search for the pupil (the top part of the above figure). Generally, the center of mass of the Sobel-transformed image could be found somewhere in the eye, so a box 800 pixels tall and 800 pixels wide was drawn around the center of mass of the Sobel transform. The Sobel transform function that was used was from the SciPy library. While this program was faster, it was still slow.

When the headset was printed and the new IR cameras were mounted, it was seen that the pupil was consistently in the same region, so the bounding box search was replaced with a static box centered in the top-center part of the image. This is significantly more efficient. Also, the images from the new camera allowed a transform function that makes the pupil clearer relative to other features to be written. These details are more thoroughly explained in the next section.

# Team

## James Liu

James focused on software design. James also helped with hardware, determining whether the camera and video were functioning properly. James took responsibility for formatting and organizing most of the writing - Conceptual Design, Standards, and the final poster and presentation. Regarding the design, James took responsibility for the design of the Solidworks 3D glasses model on the hardware side. On the software side, James was primarily responsible for the analytic algorithm, on which he worked with John.

## John Marshall

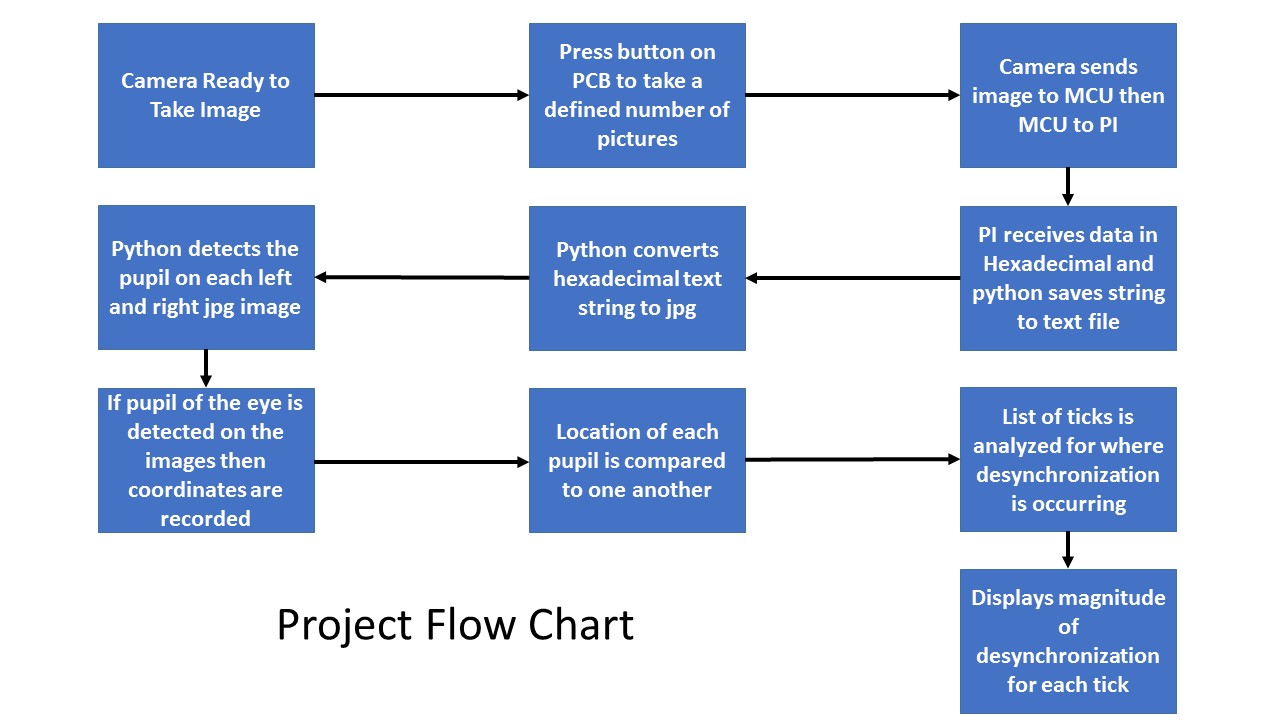
John’s primary responsibilities were the image processing algorithm, which communicates with the microcontrollers, extrapolates missing data, and converts the data to a JPG format, and the pupil detection algorithm, which takes the JPG data and determines the location of the pupil (or if there is a blink). He took on these responsibilities because of a background in image processing and recognition and significant coding experience. John’s goals in implementing the pupil detection were to reduce the problem to a computationally efficient solution that did not require machine learning, Hough or Sobel transforms, or particularly obtuse math and to format the output in a way that lent itself to the analytic program.

## Sean McCarthy

Sean’s aspect of the project was to take care of the hardware. Sean brought knowledge on how to wire and design the circuits from 501 lab. Throughout the semester, Sean learned how to code with Arduino IDE, and design custom PCBs with EagleCAD. One thing Sean had to do was setup the camera using an Arduino microcontroller. Sean had to wire the camera to function properly. Including using FTDI cables to connect each microcontroller to the Raspberry Pi. Sean then had to create code that sent commands to the camera that would take the picture, send the data of the picture, etc.

# Final Prototype

## Hardware



**Figure 3**: A flow chart mapping how the overall pipeline functions, from image capture to analysis.

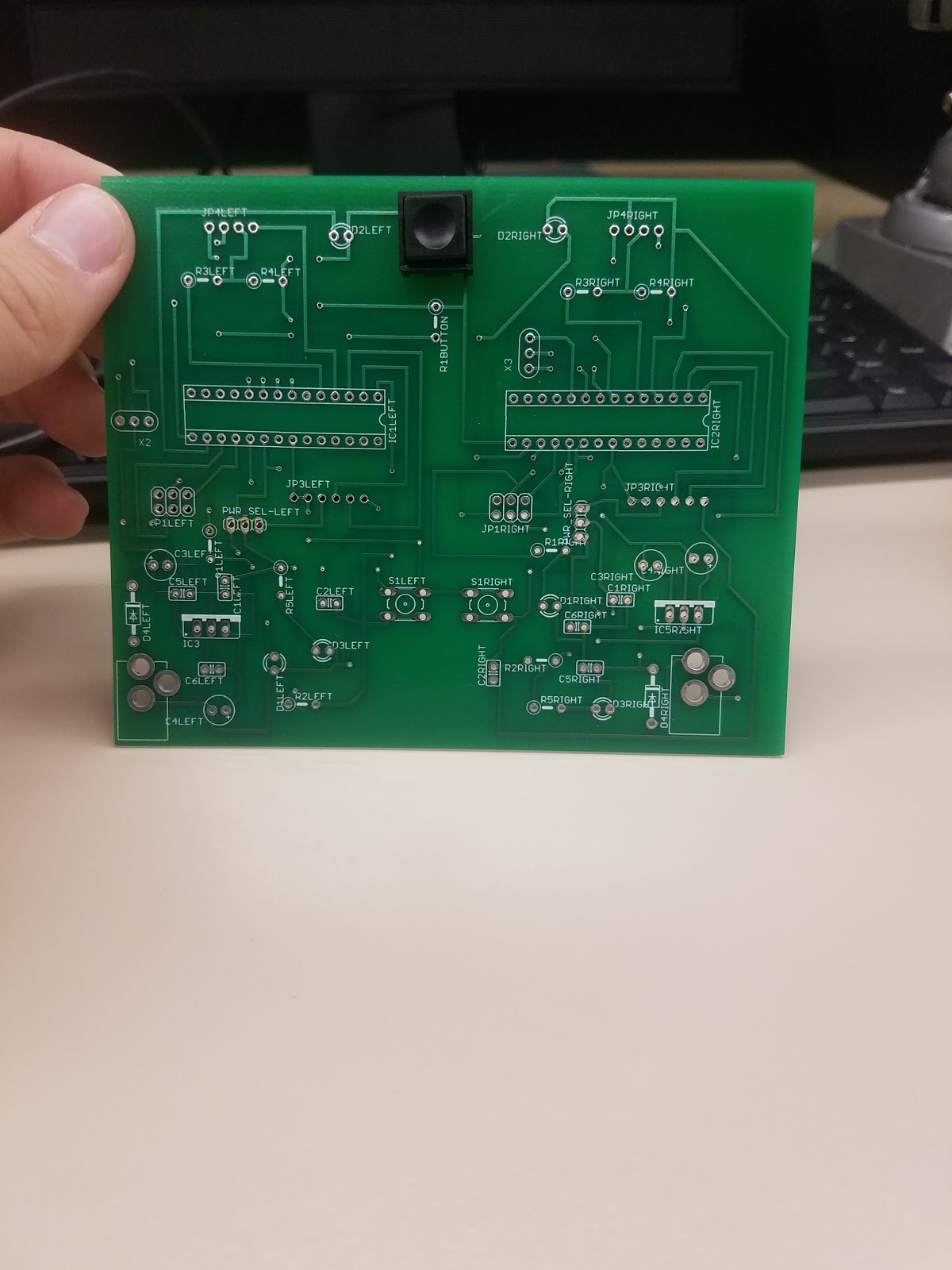
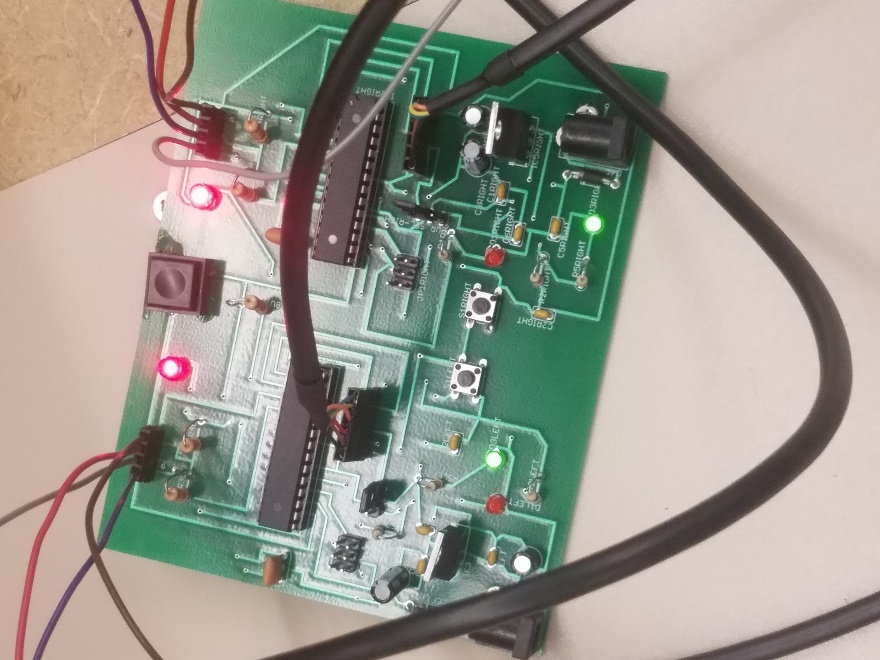
### PCB Board

When the cameras were first bought, the plan was to directly wire the cameras to the Arduino and take images. This was not the ultimate solution for a couple different reasons. The microcontroller can only send data at 5 Volts. The camera reads the incoming data at 3.3 volts. To rectify this problem, a voltage divider on the Rx data line was added into the circuit. The voltage divider was used to reduce the voltage from the camera to the microcontroller. This was a necessary change in the design of the project.

The software to control the camera was originally set as an infinite loop continuously taking pictures and sending the data through the serial port without any sort of delimiter. This would not work for a final design because the Python program would read the hex string and find predetermined text strings like “start” and “end”. Also, the hex string needed to eventually stop. Therefore, a new code was needed to end the picture and to control the cameras. The code to control the cameras was written in Arduino IDE; the current version is the file buttonforcamera2loop.ino. In this code, there are different commands sent to the camera by hexadecimal. These commands range from start picture, end picture, send the data, read the data, etc. The microcontroller sends these commands which allows the rest of the code to function. An inverse logic design was implemented with the button and wired to pin 9 of the microcontroller to control the state of the cameras.

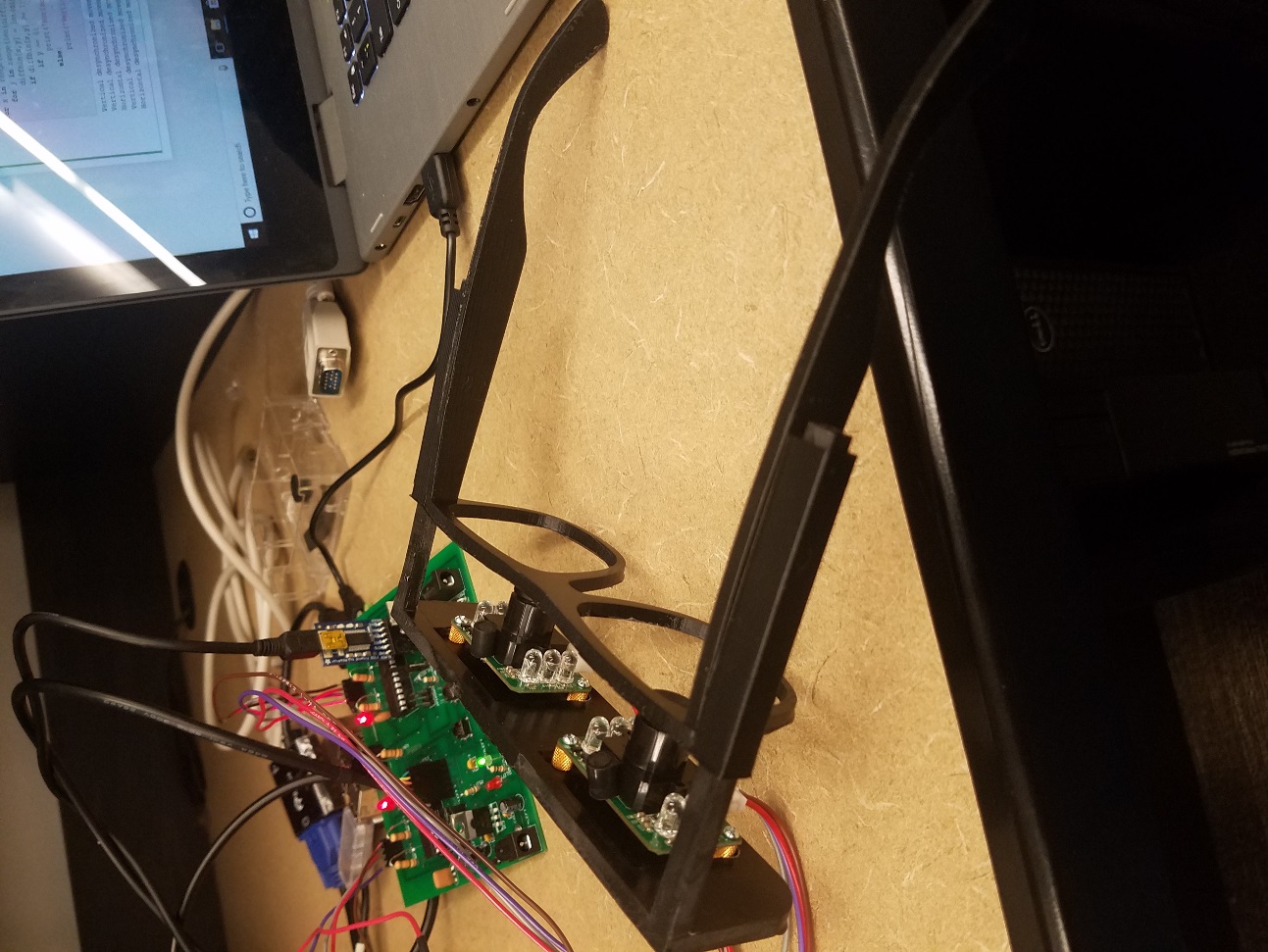
An IF statement was used in the code to read pin 9. When pin 9 had a high value it was in the IF statement which nothing happened expect output “No Pic” to the serial port. An else statement was used to control the cameras. When pin 9 was at a low value, or when the button was pressed, the code went to the else statement. This had a FOR loop that determined how many times the else statement would run. A Boolean expression was used to control the loop. It ran as long as the integer was less than a given number like three. Starting from zero it would run once. At the end of the loop one would be added to the integer. If integer I is less than 3, it will run. When integer I is equal to 3, it does not meet the requirement and exit the loop.

When the loop is running, it sends the take picture command, send data command, and read data commands. When the data starts to read and be sent through the serial port, an additional code runs. This code is necessary for the data to be read through the serial port. Then the stop picture command is sent at the end loop. Next, a rest command is sent to the camera to clear the any command and data left inside of the module. Then, depending on the Boolean expression, the loop would run again or if the conditions were not met any more the state of the microcontroller would return to IF statement.

 Two microcontrollers were needed to control each camera. Otherwise one camera would take a picture then the other would take the picture. So the design had two microcontrollers in parallel with the output of the button connected to each microcontroller. This allowed for the two serial ports to be in parallel with each other as well. Since the two microcontrollers and serial ports are in parallel the images could be taken at the same time and sent through each serial port simultaneously.

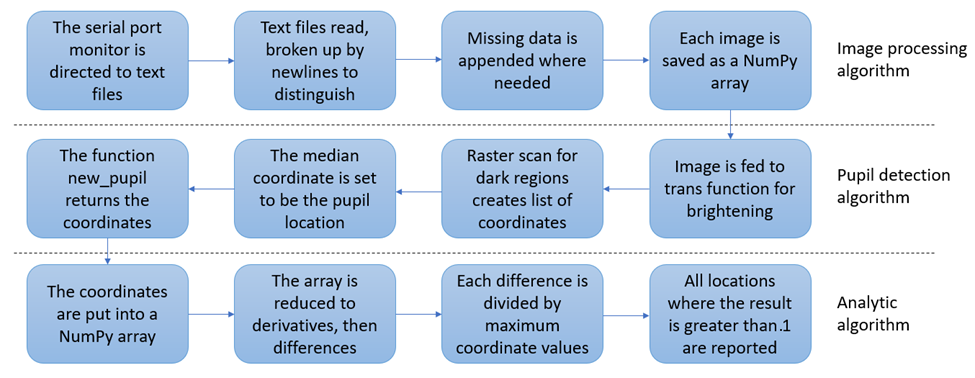
**Figure 4**: Realization of the PCB Board

### 3D Printed Headset

The headset was designed using Solidworks. The 2D outline of the frame and each leg were drawn on paper and then uploaded. The base was extruded and a supplemental camera mount was added. The specific dimensions of the frame are in accordance with the Vision Council’s Standards that are discussed in the Technical Standards and can be found on the Solidworks file attached in the ZIP.

**Figure 5**: Realization of the PCB Board

## Software



**Figure 6**: Flow Chart Summary for Software Design

### Image Processing Algorithm

While the frames are captured by the cameras and the data is sent from the camera to the microcontroller to the USB serial ports, the Python image processing program reads the serial ports’ output byte by byte, appending each to a string corresponding to each camera. This will continue until one of the serial ports goes ten seconds without sending a byte, at which point the byte reading loop will close and each string will be save to a separate text file, “sread2.txt” corresponding to the serial port USB0 and “sreadr2.txt” corresponding to the serial port USB1 (i.e., each camera writes to its own text file).

Once these text files are saved, their contents are read back into the Python code for the second part of the image processing algorithm, which delineates each frame from one another and fills in missing data. In each of the text files, the delimiter between each frame is a newline (“\n” in Python). To find the bounds of each frame, a program runs byte by byte through each text file and returns the location of each newline. If two newlines are separated by a sufficiently large number of bytes between them, e.g. 100, then all the data between them is saved to a unique position in a list. The “Start” and “End” signatures from the Arduino code are removed from each of these. This list therefore contains the hexadecimal data for each individual frame.

As a last step to make these images convertible to JPG format, missing data needs to be added. Since the data is encoded byte-by-byte in a raster pattern (left to right, top to bottom), the data missing at the end will always correspond to the lowest part of the image. Since the eye is almost always in the upper half of the image, this lower data is only important for providing a complete number of scanlines to create the JPG. Also, it is also necessarily true that the first frame will always complete. Therefore, in order to complete the missing data on any incomplete frames, the program looks at the final three bytes of each frame’s hexadecimal string. If these bytes are “FD9”, the frame completed since the unique signature that ends such a hexadecimal image is “FFFFFD9”. If these bytes are any other value, then the image is incomplete and all the data that ends the first frame is added to the end of the frame in question. For example, if the first frame completes with 300 bytes and the second remains incomplete with 250 bytes, then the last 50 bytes of the first frame are appended to the end of the second frame so that both are 300 bytes long (in reality, each frame is much longer than this).

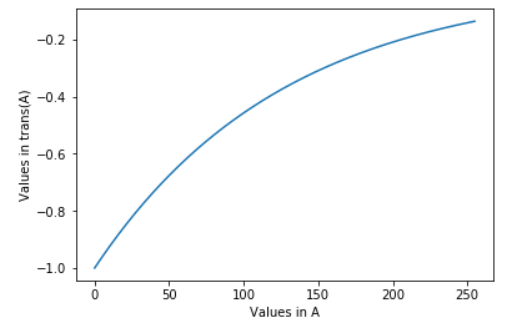
Now that all frames in the list are long enough to complete a full hexadecimal image, they are each converted out of hexadecimal using the binascii library’s unhexlify function and saved to a unique location, e.g. “sr\_0.jpg”, “sr\_1.jpg”, “sr\_2.jpg”, etc. (The hexadecimal signatures beginning and ending each string provide enough information for the unhexlify function to recognize each string as an image compatible with the JPG, BMP, or DAT formats.) If these files already exist, they’ll be overwritten. Throughout this process, for loops are used to preserve generality. Any arbitrary number of frames can be read into the code, so that only the Arduino code needs to be changed for the total number of frames to be changed.

### Pupil Detection and Analytic Algorithm

The pupil detection algorithm comprises custom three functions: trans, new\_pupil, and pupil\_from\_serial. The main function is pupil\_from\_serial, which uses new\_pupil, which in turn uses trans. Each of these functions will be described in detail.

The function trans takes a single argument, the NumPy array of the relevant frame, and returns a transformed array in which the pupil is generally darker relative to the iris and skin. This function was designed by trial and error, being inspired by the general principle of changing the relative concentration of certain brightness values with an exponential function. Assuming A is the input array, then, for element (i,k),

where min(x) is the minimum value in x, max(x) is the maximum value in x, and mean(x) is the mean value of x. The array D is what the function trans returns, i.e. D="trans(" A")" . As seen in the last line of the formula, the result is normalized between 0 and -1. This is necessary for new\_pupil, as will be shown next.



**Figure 7**: The remapped values from trans, showing how the range of pixel values 0 - 255 in A become -1 - ~-.14



**Figure 8**: The input, A, left, and the output, trans(A), middle. Notice how the shading details on the skin and sclera are greatly diminished, the iris becomes much lighter, and the pupil remains dark. The final new\_pupil result is on the right.

The function new\_pupil is what calls trans. There are two arguments for new\_pupil: the first is the input image array, and the second is a truth value called justimage, by default set to justimage=False. The first thing that new\_pupil does is call trans with the input of new\_pupil as the argument. Then, a target range of dark values is determined. In the current version of the program, this is between the minimum of the transformed image, generally -1, and that value plus .35. In other words, the values of interest are roughly the darkest third of pixel values after using trans.

Next, a region of interest is selected to throw away the dark edges along the side of the head and the shadow of the nose. Originally, this was selected to be centered on the center of mass of the image’s gradient, but after testing with the printed headset, it became clear that the eye almost always fits within the 50-60 pixels of the vertical edges, so the bounding box was always set to these values. An additional “slide” variable sets the lower horizontal bound, which is by default set to 30. This is only a problem when a significant portion of the lower part of the image is missing.

Now, a search in a raster scanning pattern is performed. The image within the bounding box is quantized to a grid of squares 20 pixels wide (this is the “step” variable). The value in each square is the median pixel value in that region. Originally, mean values were used, but this was replaced with median to compensate for glare. Notice how, in the above figure, there is glare in the pupil. Most pixels are very dark, but those within the bokeh circles of the glare are very light. These significantly increase the mean brightness of the regions within the pupil, but not the median. As the search moves in its raster scan, it records each coordinate where it found a value within the target range. When the search is complete, it analyzes the list of coordinates it found. If there are no entries in the list, new\_pupil identifies this frame as a blink and returns a coordinate value of [10000, 10000]. Otherwise, it returns the mean value for each coordinate in the list as the final pupil coordinate.

Overall, this algorithm is significantly faster than a circle Hough transform both because of the quantized steps and because it looks for dark regions, not necessarily enclosed regions, a luxury afforded by the infrared camera and the trans function. This algorithm would probably not work for a camera that takes visible-range images.

If justimage=True, instead of returning the coordinates to the rest of the code, new\_pupil plots the annotated image showing the transformed image and the pupil. This is a feature that was used for tuning, debugging, and demonstration, for example in the script pupilfromserial.py, but is not used in the main expo\_demo.py script.

Finally, pupil\_from\_serial is a function that combines the part of the image processing algorithm that creates frames from the hexadecimal serial port output and new\_pupil. The argument of pupil\_from\_serial is the serial port output file, e.g. “sread2.txt” or “sreadr2.txt” and the output is a NumPy array of coordinates for each frame. Each successive row in the output represents a different frame, ordered as they were in the serial port output. The first column is *x* coordinates and the second column is *y* coordinates. In expo\_demo.py, the pupil\_from\_serial output arrays for “sread2.txt” and “sreadr2.txt” are concatenated together horizontally to be used as the input of the analytic algorithm.

The analytic algorithm takes this input in the form of a NumPy array. The format is show below.

This data is then used to create a new array with the format

.

via simple subtraction, where is the change (in pixels) of the *x* coordinate. For example, the top-left-most element of the new array, diff[0,0] in the code, is the magnitude of the difference between the left eye’s from tick 0 to tick 1 and the right eye’s from tick 0 to tick 1. If there are substantial values in this array, then it is likely that there is desynchronized eye movement. However, a threshold needs to be established so that lower values that would naturally occur for healthy eyes do not create false positives. In order to accomplish this, each value in the array is divided by 1/10th (this was tuned by testing) the largest coordinate value in the relevant ticks and the resulting values are cast to integers for readability. If any value in the resulting array, diffbin, is greater than or equal to 1, this is printed in the report. Some of these reports are seen in Figure 10 below.

As a last step, the mean and median values for the array diffbin are found. If either of these values are 0, then it can be safely assumed that there was not significant desynchronization overall and the message “No serious desynchronization detected” closes the report. Otherwise, the mean value of diff is printed along with a message stating that there has been significant desynchronization.

# Testing, Data Analysis and Results

## Test Plan

General:

Take a set of frames live. These will be used to show the pupil detection. The set will then additionally be run through the analytic program; since the person who is being tested likely has no anomalous eye behavior, this will show the frequency and severity of false positives.

A pre-compiled fake desynchronized set will then be run through the analysis to show what the results look like for actual cases of amblyopia/strabismus and how these compare with the false positives.

The metrics that will be recorded will be the rate of false positives and, if applicable, the rate of false negatives for the fake lazy eye set. Instances where a pupil was recorded in a location outside of the bounds of the iris will also be recorded.

Specific:

Hardware/image processing testing:

• Take an arbitrary number of frames of each eye simultaneously and have the output of each corresponding serial port converted to a set of jpgs

Pupil detection testing:

• Using these frames, have the pupil coordinates return for each

• Show that blinks are properly detected

• This will be run automatically after the cameras are done taking images

Analytic program testing:

• Run the analysis for one of the sets of frames just taken to demonstrate the incidence of false positives of desynchronization (assuming the person whose eyes were used is healthy)

• Run the analysis again for a set generated beforehand that will simulate lazy eye; this will show whether there are false negatives occurring

This test plan was performed and the results are described below.

## Hardware Testing

During the soldering of the PCB, random testing occurred to make sure the microcontroller was working properly. The example code for the blink test was uploaded to the each microcontroller then ran to see if the computer would communicate with the PCB. Then, using the serial monitor on a separate computer, the PCB ran through the Arduino code. The data of the image would be sent to the serial monitor. This confirmed that the PCB was working correctly.

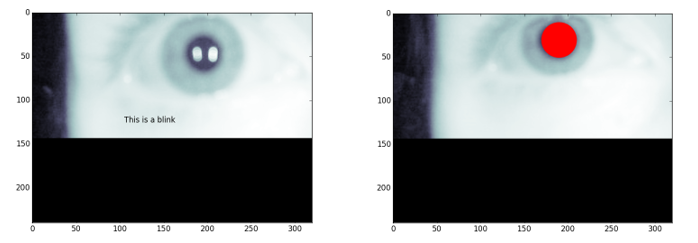
Tests were also performed to make sure that the headset could be worn stably on the head. When all the constituent pieces of hardware were confirmed to be working individually, they were combined and testing then became ensuring that the serial ports were reading correctly on the Raspberry Pi. After the locations of each serial port in the Debian file structure were correctly identified in the Python code, it was seen that the ports were read correctly.

Initially, during testing, the present byte being read would be printed to the Python shell as long as the serial ports were running. However, this was found to slow the program and prevent the images from completing, so this feature was removed.

## Software Testing

### Pupil Detection Testing

Since the only way to quantitatively measure the error of the pupil detection algorithm would be to manually identify the center of each pupil by eye, which is still subjective, we tested whether the accuracy of the pupil detection was sufficient by looking at the annotated results of images, e.g. by running them through pupilfromserial.py, and judging whether the results were sufficient.



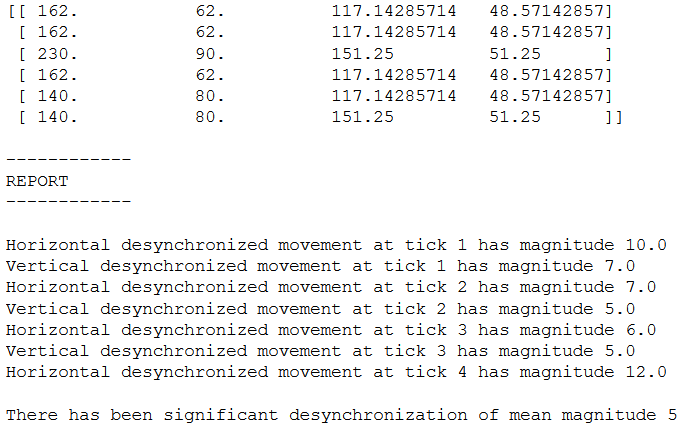
**Figure 9**: Two frames taken at the Design Expo with missing data, one with a false blink and the other with a correctly identified pupil. More of these frames are in the annotated\_frames folder

The most important aspect of the verification for the pupil detection was making sure that missing data did not result in completely inaccurate results. As shown in the left of the above figure, this sometimes does cause a problem, especially if conjunction with excessive glare on the pupil (this specific frame is described in more detail at the end of the “False Positive Testing” section). Some increasing of the prec variable in new\_pupil had to be done to ensure that this did not totally ruin the results. As can be seen in the annotated images in expo\_1 and expo\_2, missing data does not generally cause a problem. When it does, however, it can have a major effect, as explained in the “False Positive Testing” section.

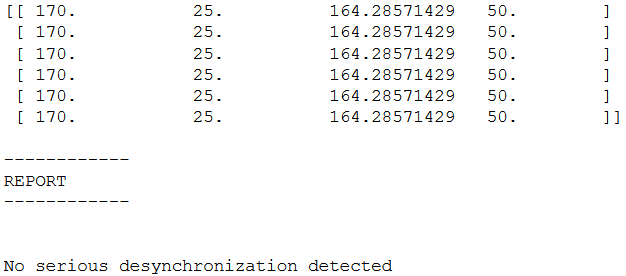
### Baseline and Fake Desynchronization Testing

Before doing false positive testing, we generated fake sets by compiling frames taken individually from the sets in the images directory. These were run using what is now the file demo\_run.py. Although this was run for randomly-constructed sets to show that desynchronization was identified for totally random data, three final baseline sets were built, each of which are in demo\_run.py (the explanation of how to run the program for each is in the README file). The first, ts\_1, is one in which the order of the frames was deliberately changed so that there would be desynchronization between ticks 1 and 3. This is the set that can be seen on the second video in the presentation. The second, ts\_2, was a set in which each eye was held constant to verify that the program was behaving as it should, producing no false data that might result from an incorrectly-written algorithm. The third, ts\_3, consisted of a left eye that was held constant and a right eye that twitched repeatedly, as might happen in nystagmus. This set was useful for showing that the normalization in the analytic algorithm was behaving properly, because the change in coordinate from the twitching was not necessarily very large, but because the overall range of values was small (between 141 and 170 for the *x* coordinate and 23 and 50 for the *y* coordinate), the normalization would still allow desynchronization to be recognized. Each report is shown below.

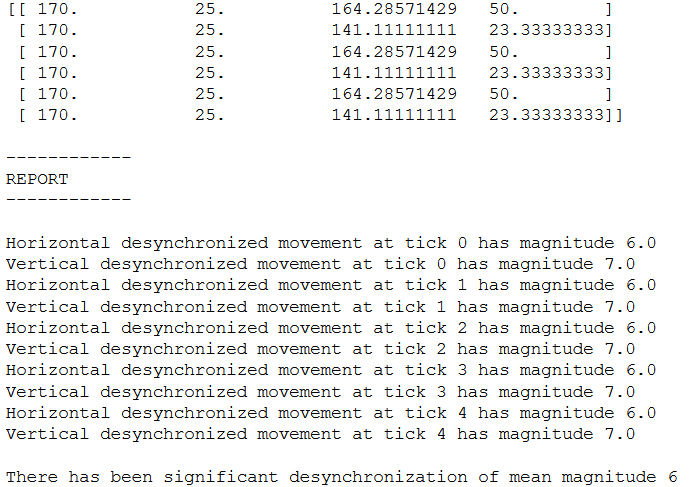
1:



2:



3:



**Figure 10**: 1: report for ts\_1, the simulated lazy eye. 2: report for ts\_2, the case of no movement. 3: report for ts\_3, the simulated nystagmus-like twitch of the left eye.

As can be seen, there was no algorithmic error on the unmoving test ts\_2. This is a validation that the program works generally as expected. For ts\_1, there were both vertical and horizontal components of desynchronization for ticks 1, 2, and 3 and only a horizontal component for tick 4. These are consistent with the behavior that would be expected from looking at the frames themselves. For ts\_3, the individual relative desynchronization magnitudes show that the periodic twitching was successfully identified.

Importantly, the fake desynchronization sets ts\_1 and ts\_3 each result in an identified overall significant desynchronization: 5 for ts\_1 and 6 for ts\_3. Compared with the results from the Design Expo, described in the next section, which almost exclusively identified no overall desynchronization, this shows that the analytic algorithm is generally able to successfully discern between significant desynchronization and normal, healthy desynchronization. While false positives for individual frames can be in excess of the 5 or 6 overall values for ts\_1 and ts\_3, it is rare for there to be any overall desynchronization due purely to false positives, as will be shown next.

### False Positive Testing

During the Design Expo, we had visitors to our booth wear the headset while we ran the expo\_demo.py script, manually recording each result. None of the individuals who agreed had known desynchronization-causing illness. The table below shows the results of these runs. The right column shows the median value of individual ticks’ relative desynchronization (e.g. “Vertical desynchronization at tick 1 has relative magnitude 7.0”) *only* counting ticks where there was such desynchronization; if there were no relative desynchronizations, the median was 0. For the other column, it is stated whether the program determined the session to have an overall desynchronization (e.g. “No serious desynchronization has been detected”).

|  |  |  |
| --- | --- | --- |
| Person | Identified overall desynchronization? | Median relative desynchronization |
| 1 | No | 7 |
| 2 | Yes | 33 |
| 3 | No | 0 |
| 4 | No | 7 |
| 5 | No | 6 |
| 6 | No | 0 |

**Table 1**: Data Collection Cohort at Senior Design EXPO

As seen in the table, all but one person had no serious desynchronization detected. In other words, the mean and median values of the data in the array diff (the normalized data) were below the threshold. This data suggests that an *overall* false positive for an entire run will happen for a minority of individuals (1 in 6 for these sessions). Assuming that this would be true for a potential user with no amblyopia or strabismus, even if they occasionally got false positives for a session, there would not appear to be an overall change. There is no reason to believe that any factor would cause an increase in false positives over time.

The relative desynchronization data, and the annotated images, demonstrate why the results for Person 2 were so unusual. The images for this run were saved in the folder annotated\_images/expo\_2. On the frame labeled im0la, the frame is clearly identified falsely as a blink. The corresponding frame for the other eye, im0ra correctly identifies the pupil. Therefore, where the left eye had coordinate roughly [100, 50], the right eye with the false blink had coordinate [10000, 10000]. It is clear why this would produce an extreme value for desynchronization (the relative value 33 is substantial; generally anything over 10 is very significant.)

The reason the frame im0la was labeled as a blink likely has to do with the large amount of missing data at the bottom of the frame (the black lower third). This caused the normalization in trans to brighten the dark portions of the pupil relative to the black region. Combined with the glare in the eye, which would probably not have been a problem without the missing data, the median value in the pupil was too relatively light and the raster search did not find the region properly. This is a sort of error that is probably unavoidable with our hardware setup, but a change in which frames involving a blink are altogether ignored would possibly resolve these problems in the future. Nonetheless, the rest of the data shows that the program worked well, at least for the people at the Design Expo. A set of frames from a different Design Expo session that worked correctly is saved in the folder annotated\_images/expo\_1 for comparison.

# Project Timeline

Project Milestone Check-Off

* + October 23: 1st Graded Check Off/Demonstration
    - Hooked up a Pi camera to Rasberry Pi and, using Python displayed the image
    - Used early Pupil detection algorithm to identify the location of the pupil in the frame
  + November 6-2nd Graded Checkoff/Demonstration
    - Designed headset
    - Had IR cameras connected to the Boarduino with Arduino code determining number of frames taken
    - Designed image processing algorithm to read in from serial ports and convert hexadecimal to JPEG format
  + November 20: 3rd Graded Checkoff/Demonstration
    - PCB designed, soldered together and working with LinkSprite cameras
    - Fixed image processing algorithm so that missing data in incomplete frames was extrapolated
    - Pupil detection was tuned to the new new infrared images
    - Early analytic algorithm was written
  + Between 3rd Checkoff and Design EXPO
    - Collected image data from a human cohort
    - Have entire hardware pipeline working
  + December 6 – Senior Design EXPO
    - Demoed program from image capture to analytic algorithm output
    - Continued collection of data for false positive testing
  + December 8-Final Demonstration
    - Demonstrated that program ran to completion successfully and provided reasonable results

# Conclusions and Future Work

Our headset and software generally accomplished the application that we hoped to implement. The design can read images from the eyes, identify the pupils, and make determinations about the relative amount of desynchronization between each eye over time. There remain limiting factors to the accuracy and applicability of the program, most importantly the baud rate of the cameras, but also the missing data problem.

In implementing this project, we learned and improved on a wide range of skills, including PCB design, Arduino IDE coding, working with serial ports, image encoding and processing, and software design. Prior to this project, the group members had limited or no knowledge of how to use CAD or control data flow on Arduinos. Much was also learned about the efficiency vs. accuracy payoff of various algorithms.

Any likely future work would fall into two categories: camera improvements and further optimization of the software. If smaller, miniaturized cameras with significantly faster data transfer could be mounted above or below the eye, instead of on a plane in front, then this would solve a few problems with our current setup. The eyes would not focus on a spatial plane so close, which would reduce the difference in the direction of each individual eye. Also, a faster baud rate would allow the overall program, i.e. expo\_demo.py, to be run much more quickly with many more frames.

Optimization of the software could be done in response to these changes. The normalization and thresholding in the analytic algorithm would likely need to be changed, for example, if the frames captured significantly faster, since the distance the eye could move between each frame would be reduced. Additionally, further work could focus on taking the existing pupil detection algorithm, which has been built to specifically work for the IR images that are taken by the current cameras, and generalize it to work for a more diverse range of images. These changes would be necessary to create a program that would work for other hardware implementations.

# References

1. “Standards.” *Standards | The Vision Council*, [www.thevisioncouncil.org/members/standards](http://www.thevisioncouncil.org/members/standards).
2. *Ethical Guidelines for Ophthalmologists: Ethical ...* <http://www.icoph.org/downloads/icoethicalcode.pdf>
3. “Human Subjects Research - IRB Home.” *American University*, www.american.edu/irb/.