Eye Tracking Autism for Mobile

University of California, Santa Cruz
Winter Quarter 2015

Winter 2015 Report

Team Members:

John-Michael Burke
Team Lead, Gaze Tracking
jburke2@ucsc.edu

John-Michael is a fifth year robotics engineering student. John-Michael will be responsible for developing the gaze tracking algorithm and optimizing the eye tracking application for the embedded system.

Nate Cancio
Eye Tracking, Data Storage
ncancio@ucsc.edu

Nate is a fourth year computer engineering student with a concentration in computer systems. In this project, Nate will be responsible for working on the eye tracker as well as data storage for the system.

Mike Madsen
Peripherals, Form Factor
mwmadsen@ucsc.edu

Mike is a fifth year bioengineering student with a concentration in bioelectronics. In this project, Mike will be responsible for integrating the peripherals of the system and engineering the form factor of the device to maximize ease-of-use.

Abstract:

The ETAM team aims to build a mobile device which can perform eye-tracking on a user and use the data to determine the where the user is looking at the screen. The device will use this functionality to conduct a test which assists in the diagnosis of autism in young children. The purpose of this project is to serve as an aid to those in the medical community who are qualified to perform such tests.

Motivation:

The inspiration for how our device will work comes from research done by Dr. Karen Pierce from the UCSD Autism Center for Excellence. She has found that observing the gaze of a child who is watching a video divided into two sections, one side with social images and the other side with geometric patterns, can aid in an early diagnosis of autism spectrum disorder.

Our goal is to create a mobile device which can perform these tests, so that other medical professions would have an intuitive physical tool that would allow them to diagnose their patients as well.

The test is not intended as being a replacement for a thorough diagnosis from a physician but rather an aid to call attention and recommend a thorough diagnosis of autism. The benefit of this would be detecting autism early in the child's life which should allow parents and teachers to prepare themselves earlier to give the proper care needed for the child.

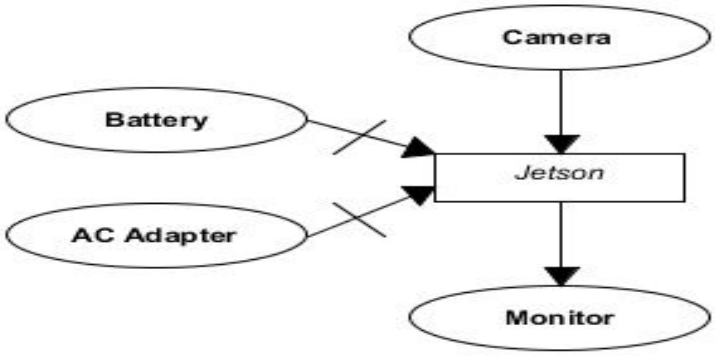
Objective:

The goal of this project is create a portable device which can track the user's eyes and map their pupils geometrically onto corresponding images on the screen. The device will be able to:

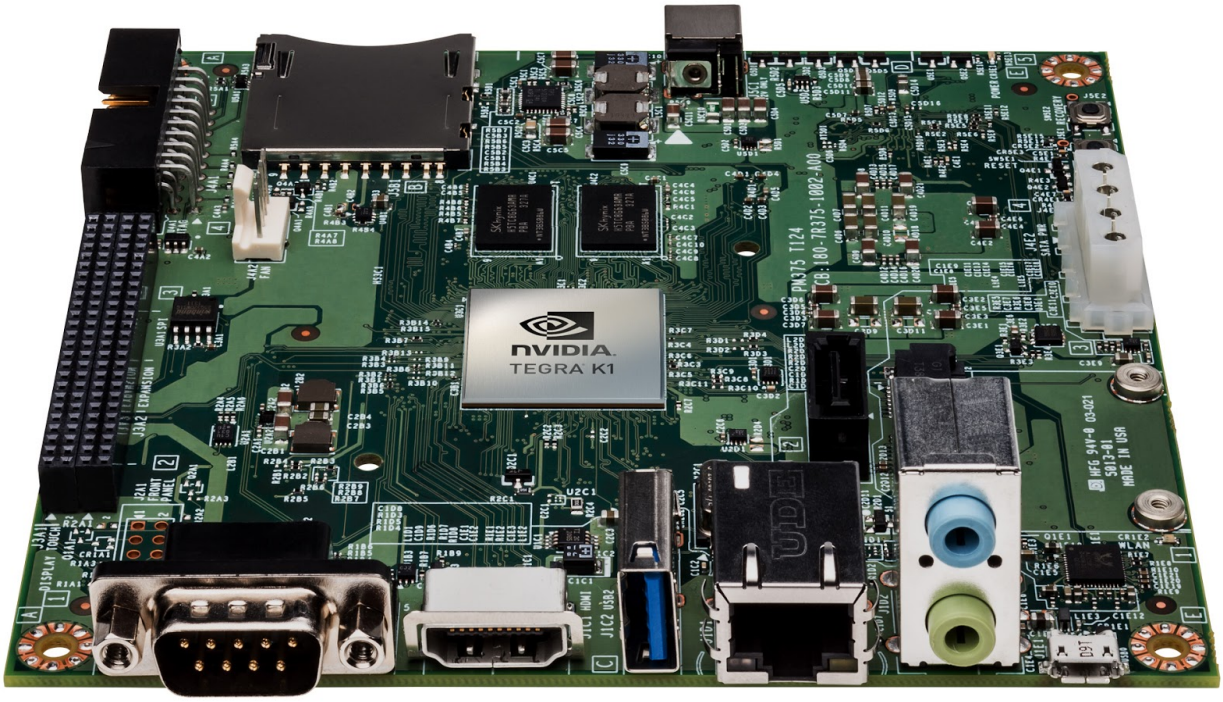
a. Calibrate itself based on the position of the user's eyes relative to the screen,
b. Play side by side videos of geometric patterns and social images,
c. Track the user's eyes in real-time,
d. Use the eye tracking data to find when and where the user was focusing on something,
e. Find the percentage of time that the user spent looking at each video, and
f. Store the data for the medical professional to analyze later.

The system utilizes the Jetson TK1 dev kit to complete the processing power intensive task of tracking a user's eyes in real time.

System:



Jetson TK1 Dev Kit:



The Jetson TK1 is a development board made by Nvidia which houses the Tegra K1 (TK1) mobile CPU/GPU. The Tegra K1 has a quad-core CPU combined with a 192 Keplar core GPU which is the first mobile processing unit to support the CUDA parallel processing language. We decided to develop for this unit because up until now most mobile processing units have been quite limited in processing power and more importantly have limited support for general purpose graphics processing (GPGP). GPGP is great for a multitude of things such as gaming but what we desire from the TK1 is the boost in performance to computer vision applications. As students we do not have years of experience in low level graphics processing languages or parallel languages such as CUDA; however, OpenCV, a widely used computer vision library, has an entire module which uses the CUDA language to accelerate common computer vision functions typically ran on the CPU. This in combination with multi-threading allows us to create a functional application which would normally be impossible to run on a single core CPU with any practical results.

Now one might question: “If the biggest constraints are processing power, then why not use a non-mobile desktop computer?” This is definitely a valid point and has in fact been done before. The problem we see with this optimal setup is versatility. We believe that with the proper training anyone in the medical field could operate this

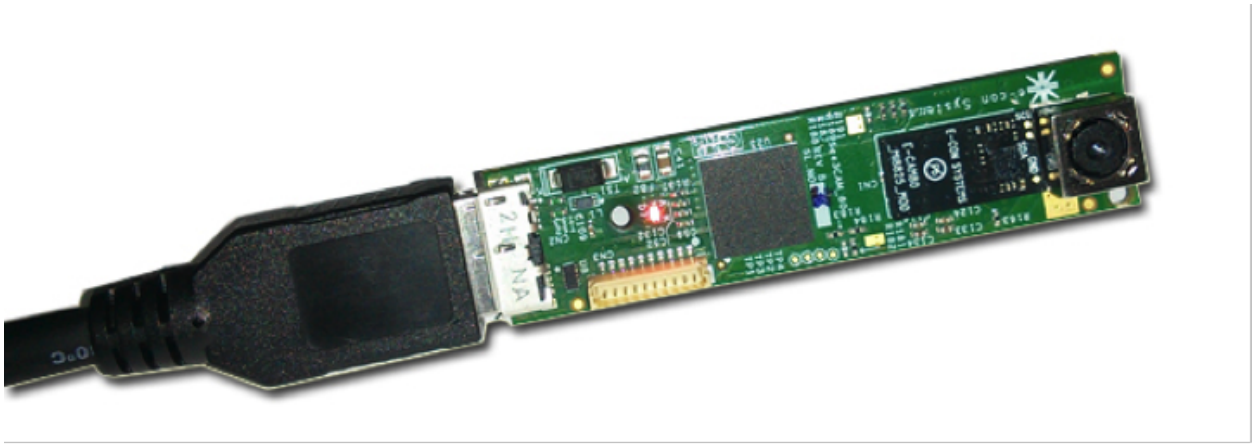
application on a mobile device. One example of a possible candidate for administering a test using this device could be a nurse at a preschool. In this case a desktop computer would not be convenient let alone affordable by most preschools but a modern tablet computer would be more reasonable. At the time we started the project, the TK1 had only been out for a little over half a year which meant the newest round of phones that came out in September would have already had their processing units solidified; however, the probability that phones and tablets coming out in future years will have the TK1 is reasonable.

Camera:

The Jetson development board advertised that it had a CSI-2 connector which would be an optimal connection for a camera because it uses four differential data lanes to send image data and can reach speeds of up to 60 frames per second for 1080p video which mirrors the performance of cameras on most phones and tablets. Unfortunately, Nvidia has yet to release CSI-2 drivers for the operating system, Linux4Tegra (L4T), they designed for the Jetson. So we had to take a different approach. Some of the other options we had were gigabit IP cameras, a GPIO camera, or a USB-3.0 camera. Gigabit IP cameras have been a staple for quite some time but can not compare with the speed a CSI-2 connection can provide. A GPIO camera could potentially work but this would require modifying the I/O voltages as well as writing our own drivers which is even worse than our situation with the CSI-2 connection. This leaves us with a USB-3.0 camera although this is actually not a bad compromise whatsoever. USB-3.0 has not been around for very long so it is typically associated with data transfer e.g. thumbdrives and external hard-drives; however, it is known for reaching extremely high data transfer rates upwards of 640 Mbps (2). This speed is quite comparable with the CSI-2 connection but it could experience a slight delay from encoding and decoding the USB format; however, this delay is more noticeable in USB-2.0 cameras than USB-3.0 such that it is almost negligible. Also there is an immense amount of support for USB cameras in Linux so it is very easy to work with.

Due to USB-3.0 being fairly new, there is a limited amount of affordable USB-3.0 cameras with respect to our budget as students. We ended up finding a camera from the company e-con Systems which was relatively affordable and was able to stream 1080p video at 30fps in a raw YUV format which sufficient for our systems needs. The camera is called the See3Cam_80. The camera has an autofocus feature which helps us reduce the amount of calibration for users at different distances from the system. The camera is also very compact so we can maintain a small form factor in our system case.

If we had a USB-3.0 camera that could produce 720p at 60fps, it would probably be better for our needs but we found that this camera is still sufficient.

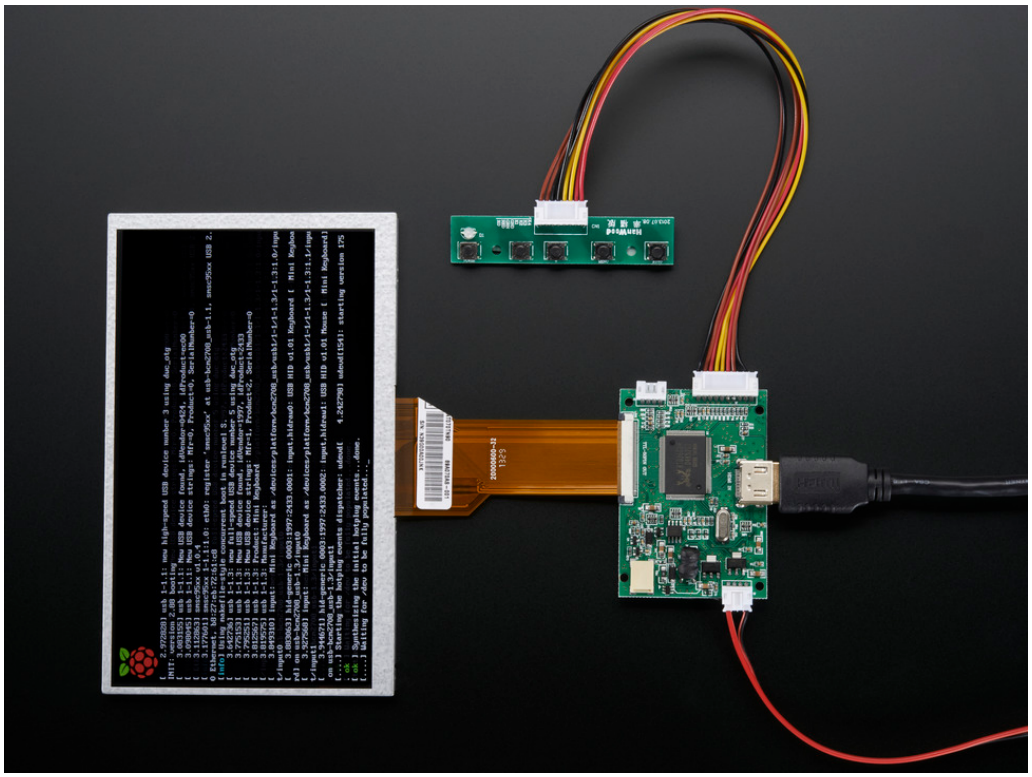


One of the issues we encountered when using this camera was that the connection from the camera circuit to the breakout board came undone. At the time, this connection was not apparent so we inquired e-con Systems to fix our camera at the cost of labor fees. We discovered that the company did not have the most fantastic customer service because they never responded to our request. Fortunately, we eventually found that the connection of the camera was ever so slightly detached from the board and just needed to be firmly pressed to lock into place. In the meantime, we had to find a quick replacement until we found a solution to our problem. So we investigated another camera which was only USB-2.0 but advertised it could reach speeds of 60 fps with 720p video using MJPEG compression. The camera was from an unknown Chinese manufacturer called ELP. The camera was called the ELP-USBFHD01M-L21.

This turned us toward the possibility of handling compressed video in order to reach the speeds of our previous camera. The big difference would be that instead of having high quality uncompressed video in the YUV format we would instead have deal with a degraded

quality video in the MJPEG format. In addition, we had to recompile our OpenCV build to support MJPEG decoding which was a hassle. After setting everything up we found that the camera was able to work in a pinch but could not compare to quality of the original camera. Our goal is to have to integrate this camera into a secondary case to use as a backup. Fortunately, after a little headache we eventually fixed the original camera so this problem dissolved and we were able to get back to the status quo.

Display:



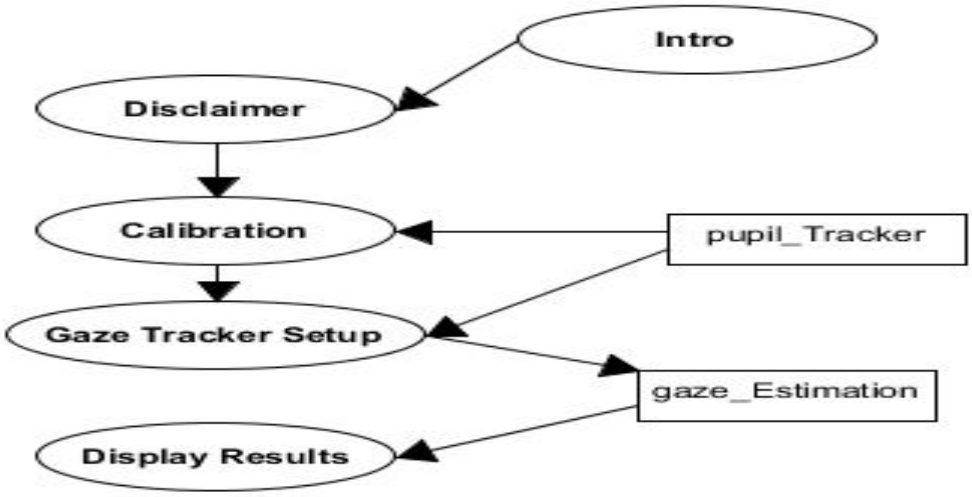
Originally our plan was to use a 5 inch monitor so that our system would be small and portable, much like a mobile phone. However, using a 5 inch monitor meant that our videos would be half of that screen size, and could be hard for the user to see. So instead we went for a more tablet sized option, a 7 inch monitor from Adafruit.com called the AT070TN94. The screen has WVGA 800x480 resolution with a thin film transistor (TFT) display. The monitor uses HDMI output and is powered by USB at 5 V and 500 mA. We will most likely not use a USB port to power it since we want to save our USB port for the camera, and instead we would power it with a regulator from the Jetson's power supply. The little PCB with the buttons at the top is a menu control system, letting you adjust brightness, color, and contrast.

So far we have been having a little issue with setting up the monitor with the Jetson because 800x480 is not a native screen resolution for the Jetson so we need to modify some of the system files in order to allow this screen size. This has proven to be somewhat difficult and requires us to ssh into the Jetson and debug the screen in order to modify the system files and test if the modifications worked. Otherwise we would have to go back and forth between a working and non-working monitor for debugging which would require us to reset the Jetson in order for it to recognize the new different monitor each time. So long story short, this has been a slight hassle.

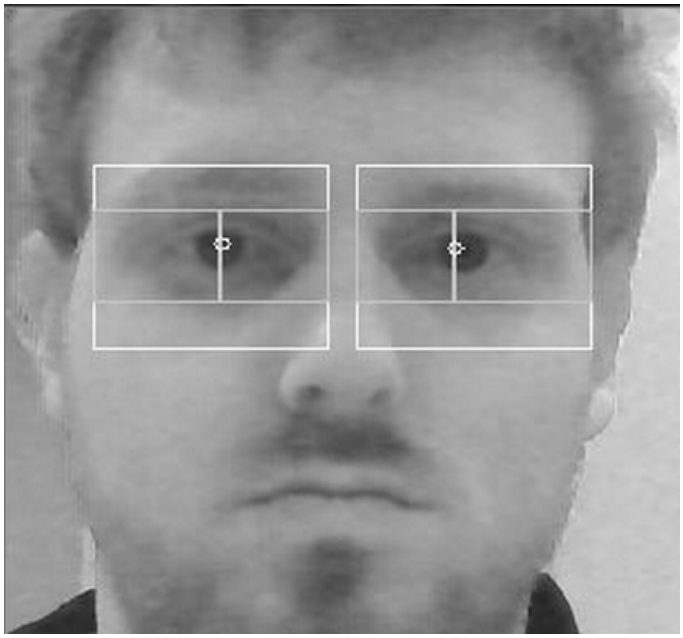
Battery:

Shown above is the battery we plan on purchasing for our system. It can supply 2200 mAh at an average of 11.1V. We've done research to find out the typical wattage of computer vision tasks on the Jetson, and found that they range from 2 to 11.5 Watts (including at least 1.6 Watts to power rest of the board). If our program and system reached that higher bound of wattage, This battery could power it for 2 hours straight of continuous eye and gaze tracking. Since using our device won't take longer than a few minutes at a time, we find that this kind of battery is promising. The only downside is that in addition to buying the battery pack, we will have to buy a clunky battery charger, which is not ideal. Li-ion batteries are also being investigated, however the ones that we've seen have similarly bulky battery chargers.

Software:



Eye Tracking:



After researching the state of the art of pupil tracking methods, we chose the Timm and Barth algorithm. This algorithm ranked among the top three in accuracy across a range of error tolerances. The big reason why we picked this algorithm for our system is because it was one of the only algorithms in which we could find an open source implementation. This implementation is called “eyeLike” and was coded by Tristan Hume.
EyeLike begins by capturing a frame from the webcam into a matrix. This frame is then passed onto a function that splits it into its three RGB channels. Traditionally, face detection algorithms convert a frame into grayscale, however in this implementation the frame is converted to its red channel. Once we have the red channel, the face is detected on the frame using the

Haar Cascades method. If the face is not found, a new frame is extracted and the process is repeated.

Once the face is found, the process for finding the eyes and eye center begins. The general left and right eye regions are first found using the face width and height values. Next, we look at the left eye region and right eye region to find the left and right eye centers respectively. First, the X and Y gradients are calculated. Then the gradients are normalized and thresholded. A blurred and inverted image of the eye region is created for weighting. The eye center is found by computing the max gradient of the eye region. In other words, the estimated eye center is found when the X and Y gradients are equal to zero.

Optimization:

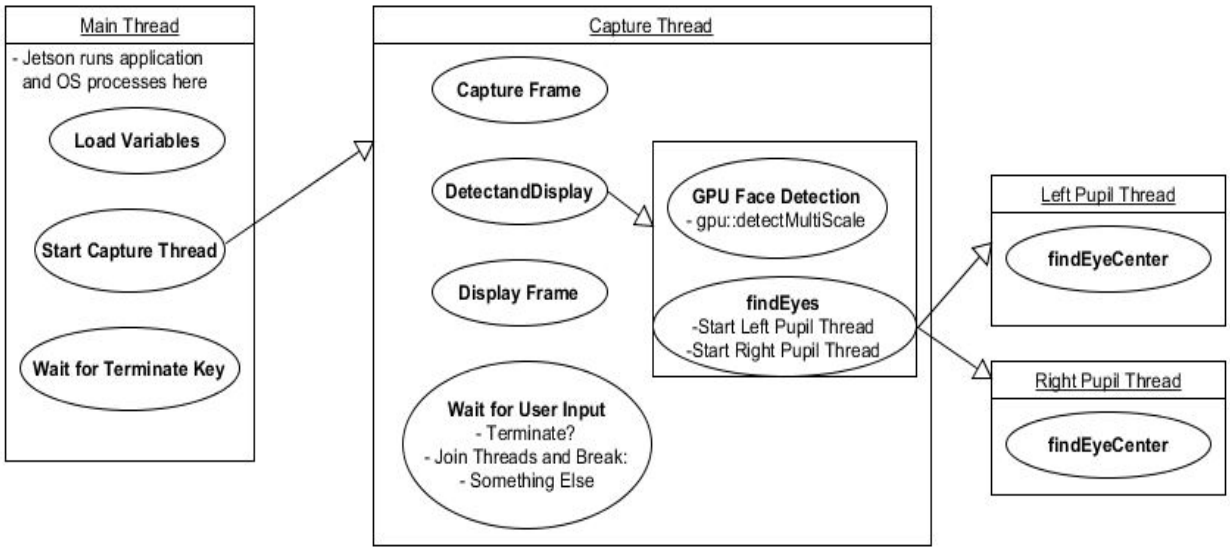
Originally, we were using Nvidia's prebuilt binaries of OpenCV4Tegra which has a few multithreaded and NEON optimizations of common OpenCV functions. This was great for quickly setting up OpenCV in order to compile the eyeLike code and begin the optimization process. The first step in the optimization process was to convert the eyeLike code to take advantage of the GPU acceleration using OpenCV's gpu module.

In order to use gpu functions you must take the camera frame stored on the CPU and download it to the GPU as a GpuMat. After this we would flip the frame and then separate the frame into the three RGB channels so that we could use the red channel for processing. Next we run the gpu detectMultiScale function for performing the face detection. Up until this point everything was an exact replica of the original eyeLIke code but instead using gpu accelerated functions and we found a significant boost in performance here. However after this point we encountered a significant obstacle. From this point on, the portion of the eyeLike code for pupil detection does many pixel by pixel arithmetic operations. Unfortunately, OpenCV does not allow pixel access for GpuMats, so we would need to use lower level CUDA code to accelerate these processes. There are a few functions throughout the pupil detection process that could be gpu accelerated but they are mixed throughout the pixel by pixel operations and would require a download and upload from and to CPU memory, respectively. Although we are still experimenting with the pros and cons that this would have, we have so far found that the overhead this causes is not worth it. So once we finish the face detection, we upload the resulting data back to the CPU and let the rest of the pupil detection code run normally. The boost in performance from the gpu face detection is definitely noticeable but we found that the majority of the heavy computing occurs in the pupil detection portion of the code. When evaluating the performance of the running application with these fixes, we found that application only runs on one of the TK1's four

CPU cores. This was also observed when running the code without any gpu acceleration. So the next step in optimizing our code was to split the computing between the four cores.

After a fair amount of researching we were steered in the direction of multi-threading. We quickly found that OpenCV4Tegra was not built to support common multi-threading libraries so we ended up building OpenCV from the linux binaries ourselves in order to support some multi-threading libraries. We ended up using the thread library that comes with the standard C++11 compiler because it was easy to dive into and quickly mock up a prototype. Another thing we found in our researching was more specific to multi-threading with OpenCV. It is recommended to run your image processing functions on a different thread than the thread used for grabbing frames from the camera and displaying them to the screen. This seems to be extremely effective for embedded systems especially if a buffer is used to hold new frames that are ready in the cameras frame buffer when the image processing thread is not ready for them yet. This is useful because running OpenCV's function for reading the next frame might be called more than what is necessary and wasting processing power.

We tried to segment the eyeLike code into four threads based on our research as well as what we observed to be the most CPU taxing portions of the code. The first thread is going to be shared among many various applications that run in the background of our operating system so we leave this for only the main thread. The main thread will start the capture thread which will perform the grabbing of frames, the face detection, and lastly the displaying of image results. Within this thread we run two threads in parallel because they are independent of each but are both required by the application, the right and left findEyeCenter functions. We first wait for the left findEyeCenter to finish and store the results and then afterward we wait for the right findEyeCenter function to finish and store the results. Then we return to the Capture thread, optionally display the results for debugging purposes, and then repeat this process. Here is a high-level overview of the multi-threaded code.



Although our multi-threading setup is still a work in progress, we have observed a significant boost using this methodology. There are still a few things we need to investigate such as integrating a buffer for ready frames while processing the previous frame and better organization of the four threads to run on all four cores the most efficiently.

The final attempt of optimizing the system thus far was stripping the operating system to prevent random processing heavy applications from running in the background and migrating to a more lightweight operating system. Originally, we were using a modified distribution of Nvidia's Linux4Tegra called the Grinch which used Ubuntu. But we ended up removing Ubuntu and switching over to Xfce which is a version of Debian Linux. It is known for having a very lightweight desktop gui. In addition to this, we removed an assortment of random applications that we felt were unnecessary for our overall system. All in all the increase in performance was about 3% on one of the CPU cores which is not a whole lot but it is not completely negligible.

Gaze Tracking:

The gaze estimation algorithm we have made is basic prototype and needs to be reworked. The first goal is to get the gaze tracking working with our method of prior calibration and then later on we will need to rework this gaze tracking to work dynamically without calibration which will be fairly difficult.

First thing we do is average all of the pupil locations measured over time at the four corners from the calibration stage. Then we use these averaged pupil locations to create an average between the left and right pupil for each corner. With this we can now create a bounding box of extremes for our gaze estimation. Next we find the width and

height of our extremes bounding box and create a ratio between this box and the size of our screen which is 800x480. Next we show our test screen for the gaze tracking and overlay a red circle with the modified pixel location which should theoretically show a rough estimate of where you are looking. Unfortunately, we have been having some difficulty with the creation of a reasonable bounding box. What we discovered is that the face detection portion of the pupil tracker will frequently get a little bigger or smaller even with your head completely still and the algorithm scales the detected face to compute the pupil locations. What this means is that in one frame the pupil could be detected at one spot and then the next frame it could be detected in a slightly different pixel without any change from the user. What we have done to circumvent this in the interim is to just use the face detection one time and then to keep our heads extremely still however this far from acceptable. We have one idea for how we will overcome this but we have not tested it out yet.

To overcome this obstacle we propose that we come up with a method for centering all newly detected faces between frames over some original master detected face size. What this will do is keep the pixel reference constant throughout multiple frames. We believe that this could fix our problem with incorrect pixel locations without having to disable the constant face detection. So we will keep the algorithm as close to the original method as possible and we believe this method would not be as “hacky” as our current fix to the problem.

User Interface:

Our system is intended to be handled by a physician, then passed on to the parent who then guides their child throughout the rest of the process. The user interface of our application is intended to be simple and easy to use. Currently there are two buttons to navigate the application: forward and backward. Depending on how we implement our database, we may need to add more buttons for user input next quarter.

The application begins with our logo screen. The next screen displays directions for the user stating to keep their head still. Along with the directions, there is a disclaimer that our test is by no means a final diagnosis of autism, but simply a possible indication of symptoms of the disorder.

Next is the calibration stage in which the user looks at each corner of the screen in order for the system to obtain the boundaries of the user’s eyes. This data is used for the gaze-tracking portion of our system. A goal we would like to achieve is to be able to estimate the user’s gaze without the need of a calibration step.

Next, a 50/50 video will play on the screen where one half consists of humans dancing and interacting, while the other half consists of geometric shapes and patterns moving on the screen. This video will last 1 minute. After the video has been watched

by the user, the gaze-tracking algorithm will take the eye data to determine where on the screen they were looking, then save the test results in a database within the device. The physician then will be able to go back and review the results of previous tests.

Currently in our application, we have a single image with the estimated gaze overlay after the calibration is done. The image we chose has a variety of objects of different sizes spread around the screen. We chose to display this image because it made it convenient for the user to be able to tell others where they are looking on the screen to check for accuracy. Below is the actual image used.

Credit: Andrew B. Meyers

Data Storage:

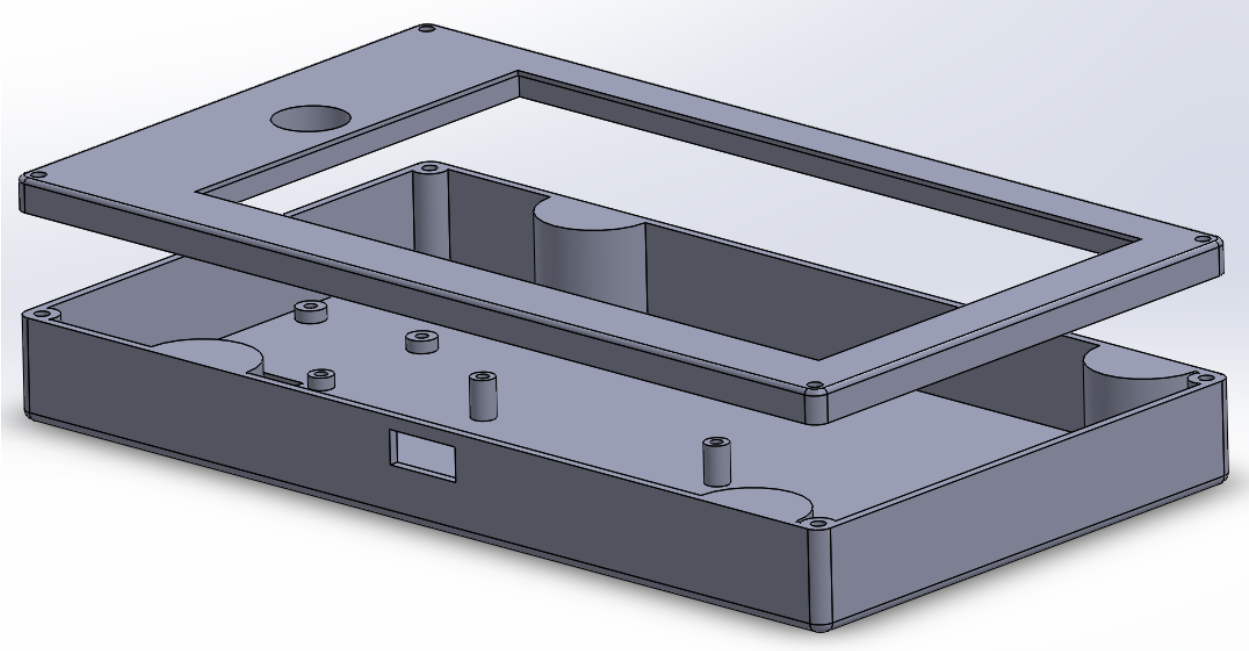
After the user watches the video, we plan to save the test results in a database on the device implemented with SQLite. SQLite is a software library that uses a

self-contained, serverless database engine. We believe that this is the best choice for our database because we wanted something that could store information on the device locally and can be easy to access.

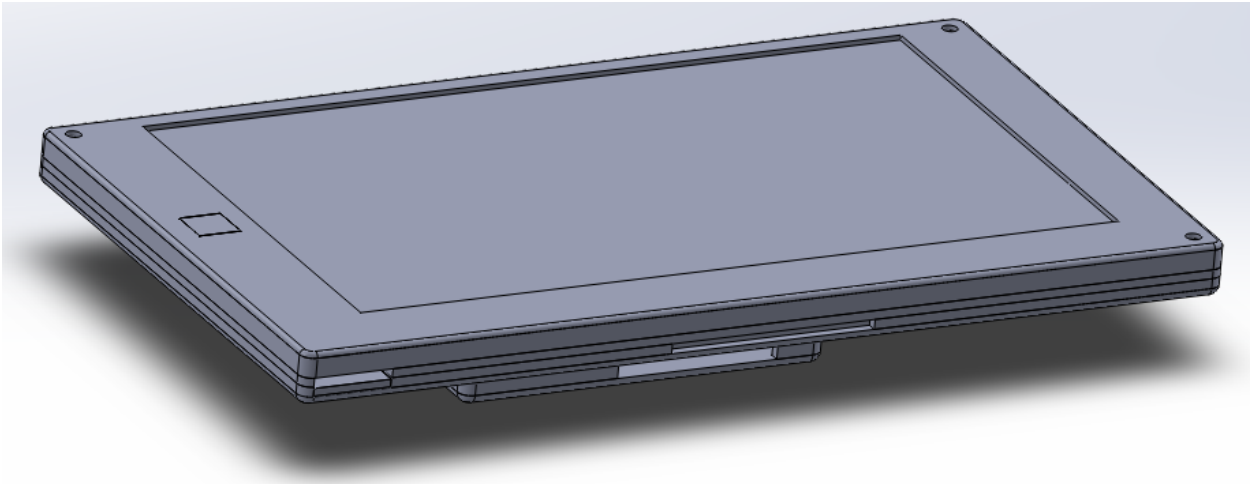
Ideally we would like our system to be able to integrate seamlessly into a medical facility in terms of how they implement their databases for storing information about their patients. However, this may not be possible to do at this stage of the project due to medical facilities using privatized software, which makes it difficult to create a way for our system to interface with their software.

Aside from the SQLite database, we may also decide to implement SD card functionality as a means of removable storage.

Form Factor:



The form factor of our case is an important part of the design process. We want users of our system to be comfortable using it, so the way they use it has to be familiar to using a mobile device such as a tablet pc. The case is made for the purpose of housing the monitor, it's modules, and the camera. The reason it doesn't house the entire device is because the development has dimensions that are not ideal for make a case around it. It is a 5 inch by 5 inch square board, with a fan on it and an audio port that juts out in an awkward way. In addition to that, the battery would also take up significant space. That is not to say that we won't integrate everything into one device in the end, especially with our original camera working again (smaller dimensions than the usb 2.0 camera), but the main goal for now is to have a prototypical case which we can perform tests with, that would be similar to using our end product.



The design above is one of the previous iterations of the case, made for the See3Cam_80. When the See3Cam_80 temporarily broke, we had to make a case that a camera with completely different dimensions, notably the width and height of it. This obvious when comparing the two cases shown, as the one with the smaller also has a thinner overall case. If we are to put the Jetson and the battery into one case, we will almost definitely use the USB 3.0 camera (not to mention its better specs).

Other design ideas that we toyed around with was to have a stand for the case, attached to a base which holds the Jetson and battery in place.

Conclusion:

We researched many state of the art eye tracking and gaze estimation algorithms and learned how they are gauged in terms of performance. We then implemented a pre-existing pupil tracker using the Timm and Barth algorithm onto the Jetson TK1. We optimized the algorithm using gpu acceleration and cpu multi-threading and achieved better performance than the algorithm compiled as it was originally coded. We began development of the application we will use for conducting the autism-test and implemented a framework for allowing us to rapidly prototype a gaze tracker. Then we developed a gaze tracker and incorporated it into our current application. The gaze tracker was far from optimal but we have isolated the problem and proposed adequate changes which we believe will solve the issue. We have experienced a few speedbumps and setbacks but we are still on track for completing our originally proposed project next quarter.

References:

1 http://images.anandtech.com/doci/7905/Jetson_TK1.jpg

2 http://www.pcworld.com/article/2360306/usb-3-0-speed-real-and-imagined.html
3 http://www.pressebox.de/pressemitteilung/vitis-pr/e-con-Systems-praesentiert-Full-HD-Autofokuskamera-mit-8-MP-und-USB-30/boxid/618719
4 http://www.amazon.com/ELP-Driver-Camera-Module-ELP-USBFHD01M-L21/dp/B00KA7WSSU
5 http://www.adafruit.com/products/1933
6 http://www.hobbyking.com/hobbyking/store/uh_viewItem.asp?idProduct=15019