User Information Augmentation. Vision Based Information Delivery System

Final Report

Group: sddec19-14

Client: Radek Kornicki

Advisor: Aleksandar Dogandžić

Team members:
Omar Abbas
Jonah Bartz
Aaron Michael
Dennis Xu

Team email: sddec19-14@iastate.edu

Team Website: http://sddec19-14.sd.ece.iastate.edu

12/10/2019

Table of Contents

List of Figures and Definitions: **2**

Project Design **3**
1.1 Acknowledgement 3
1.2 Problem Statement 3

Implementation details **5**
2.1. Tobii Eyetracker 6
2.2. Object Detection 7
2.3. Graphical User Interface 8
2.4. Interpreter **8**
2.5. Limitations and Challenges 9

Testing **10**

Related Work **11**

Appendix **12**
Appendix I - Operation Manual 12
Required Equipment 12
Hardware Setup 12
System Setup 12
3.1. Ethernet Setup 12
3.2. Projector Setup 13
3.3. Eye Tracker Setup 13
4. Software Setup 13
4.1. Windows Software Setup 13
4.2. Linux Software Setup 13
5. Running 13
6. Expanding on Implementation 13
Appendix II - Alternative Designs 14
Alternative Heads Up Display Designs 14
Alternatives to Using a Laptop as in Intermediary Device 15
Alternatives to Object Detection 15
Appendix III - Other Considerations 15
1. Jetson TX2 Development 15
2. The Sunk Cost Fallacy 15

List of Figures and Definitions:

List of Figures

Figure 1: Component Diagram
Figure 2: Data Flow Diagram
Figure 3: Tobii Eyetracker Setup Screen
Figure 4: TinyYolo Object Detection
Figure 5: Coordinate Mapping

List of Definitions:

HUD - Heads up Display
GUI - Graphical User Interface
HMI - Human Machine Interface/Interaction
NFR - Non-functional requirement
FR - Functional Requirement
UART - Universal Asynchronous Receiver/Transmitter
USB - Universal Serial Bus
TobiiSDK - Software development kit used for the tobii eye tracker 4c
GazeSDK - Software development kit used for the Android devices use with the tobii eye tracker 4c
CPU - Central Processing Unit
GPU - Graphics Processing Unit
JSON - JavaScript Object Notation
YOLO - You Only Look Once
CUDA - Computer Unified Device ArchitectureARM Processor - An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM) (Bigelow 1)
X86 processor - A CISC (complex instruction set computer) based computer architecture that was developed by Intel and AMD
ROS wrapper - Robotic operating system used for drivers
GPIO - General purpose Input Output
SSH - Secure shell protocol used for communicating with the shell of machine remotely

1. Project Design

1.1 Acknowledgement

We would like to first thank Mr. Radek Kornicki for providing supplies and guidance for the project. He has been providing us with all of the materials that we need to pursue this project. We also want to thank Dr. Aleksandar Dogandzic for supporting us with advice about how to develop our project, links to relevant literature on how to move forward in the project, and encouragement along the way. This experience has been made possible to us thanks to the Iowa State University ECPE Program and its cooperation with the enterprise corporation Danfoss Power Solutions.

1.2 Problem Statement

There is a lot of danger when it comes to operating heavy machinery. These vehicles can do a lot of damage to people and property if not properly used. In many cases, the user has a hard time knowing what is happening around them. This, in conjunction with a distracted user, can provide a very dangerous environment to work in. Mr. Radek Kornicki from Danfoss tasked the team with determining the viability of an augmented reality system in improving safety when operating heavy machinery.

Mr. Kornicki tasked us with creating a heads-up-display (HUD) system is to increase safety. The system seeks to show visuals about the environment while the user is operating heavy machinery. To accomplish this, the windshield of a vehicle was converted into a HUD. This HUD gives information to the user about potentially dangerous situations that they might be in. This system increases the awareness of its user to the objects that are in front of them at any given time. Where much has been done to achieve this task, our client merely was looking for a proof of concept, and groundwork for later projects.

1.3 Operational Environment

Our client The Danfoss Group, is known for manufacturing products and providing services used in powering heavy machinery, solar enginery refinement, food preservation and much more. Our embedded system may be used in any of these branches and given their agricultural focus we may safely assume that this can be put inside a vehicle that is moving in a muddy terrain or other less than ideal conditions. Hence the actual module itself should not be exposed but the cameras monitoring around the proximity of such a vehicle may in fact be affected. In our design the camera(s) will be separate and if this is the case, they should not be exposed to extreme heat, wet or other hazardous conditions for cameras.

1.4 Intended Users and Uses

The intended users for our solution are heavy machinery operators(like combine harvesters, cranes, and bulldozers). The images we draw onto the screen must also be clear and concise. This is because our product is targeting people in a busy work environment and anything

that hinders a driver's ability to see can lead to deadly situations. The intended use for our solution is improving safety when someone is operating heavy machinery by highlighting relevant data such as a person in front of the machine. The solution needs to be fairly generic so it can be put into a lot of different machines without too much modification.

1.5 Design Overview

Our solution makes use of the hardware the client initially provided. In order to make an HMI (Human Machine Interface) system that interacts with human eyesight. The system uses the Tobii Eye Tracker 4C and a camera for object detection system. In this case, object detection is provided by a Python/C++ based image analysis library called TinyYolo. For data projection, the use of projection film that is compliant with the Iowa Department of Transportation standards was added to keep the amount of light going through the screen greater than 70%. Voltage input is required to power the module is 5.5 V at minimum and will be through means of an electrical outlet for the time being. The entire system is tied together on the Nvidia Jetson TX2, an ARM based platform.

Functional requirements of the module are:

- Identify important objects on the road(signs, people)
- Track user's eyes in the vehicle
- Project collected data in a heads up display style
- Track eyesight over a large visual field

Non-functional requirements are:

- Object detection and eye tracking to be done simultaneously in real time
- Object detection is done with an acceptable level of accuracy
- Fast, reliable and clear HUD capable of keeping up with user while they do the task at hand
- A user-friendly calibration session for each user profile as pupils may differ due to different socioeconomic traits.

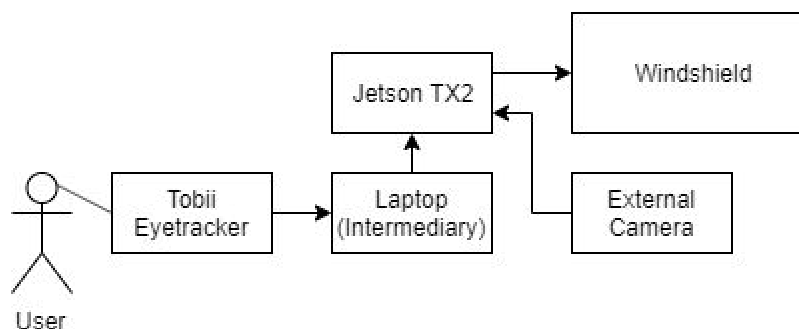

Figure 1: Component Diagram

An overview of how components are connected in the system is shown above. The Tobii Eyetracker 4C is connected to a laptop. The laptop communicates with the Jetson TX2 via an ethernet cord. Data is also collected from an external camera which connects to the Jetson. A projector takes the desired heads up display and displays it onto a windshield. The technical details

of data movement and a more in depth discussion of design decisions will be covered in the implementation section.

2. Implementation details

The implementation of this project was purely through modular design. With the given set of tools, this project aims to answer the following: Can eye tracking and Object detection work together? What would be the outcome? Our client wanted to ensure safety on the field during the use of heavy machinery, by guaranteeing that vehicle operators are aware of all of their surroundings. This project allowed our client to see if a system such as the one designed would be feasible.

Figure 2 shows the flow of data in the system. This allowed the team to isolate each part of the system and implement it independently. This sped up the implementation of the system to an extent as some aspects of the system were implemented in parallel while others proved to be bottlenecks in the project.

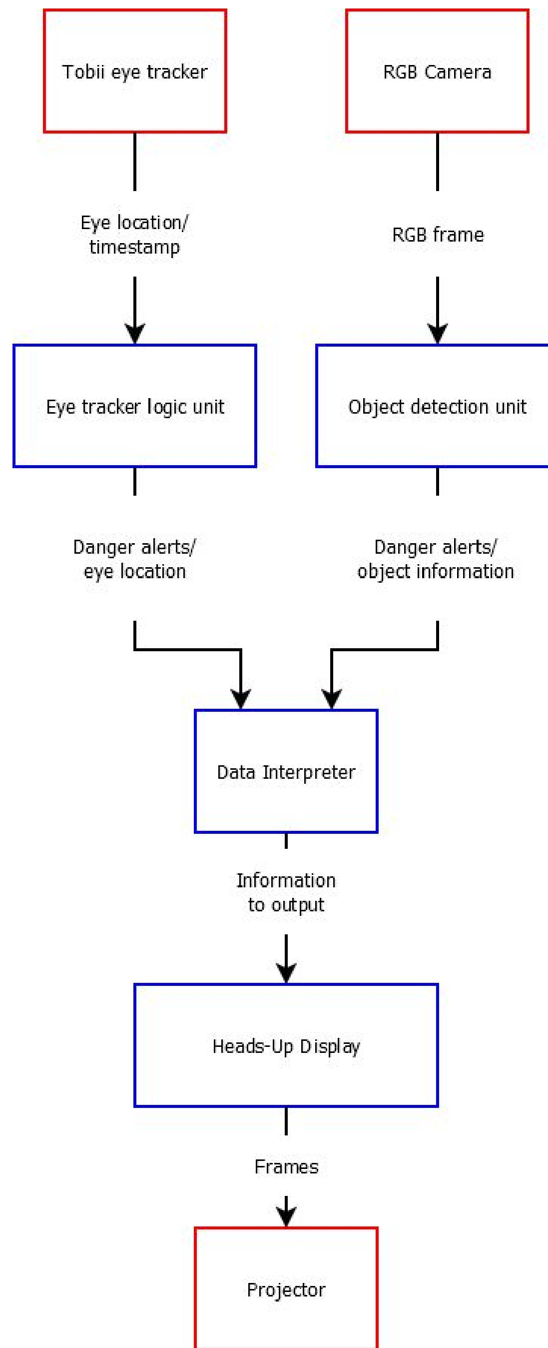

Figure 2: Data Flow Diagram

2.1. Tobii Eyetracker

The Tobii eye tracker is centered in front of the area being monitored. First, the Tobii Eyetracker is connected to a laptop via USB and the image seen if Figure 3 is projected onto the windshield. Then the Tobii Eyetracker is placed in the center of the windshield, and the eye tracker is sized to the screen by adjusting the lines on the screen which are seen in the figure below. This way, the system is able to monitor the entire area of the projection film.

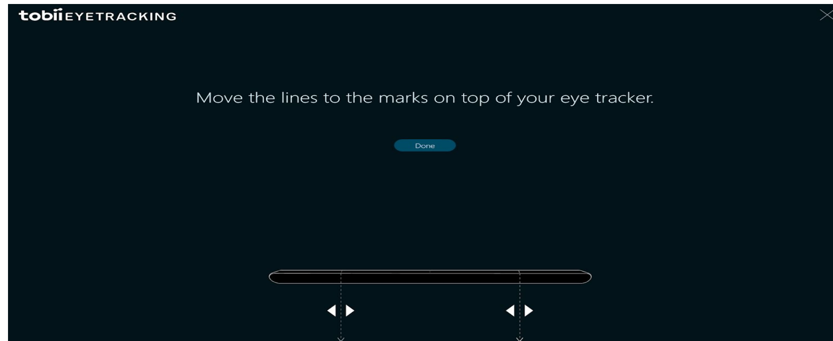

Figure 3: Tobii Eyetracker Setup Screen

The Data that is sent from the eye tracker code is a comma-separated string. This first parameter is the time that the measurement was taken. The second and third value is the horizontal and vertical coordinates respectively. This data seemed to be slightly inaccurate but worked fairly well for the purposes of this project. Hence, the first step of our implementation, which is capturing eye information can be considered to meet the functional requirements of the system.

2.2. Object Detection

The second part of implementation comes from the object detector. In this case, it is a combination of the machine learning API Yolo (You only look once) and OpenCV running on the Jetson TX2 in conjunction with the latest CUDA (Computer Unified Device Architecture) that allows for parallel use of the GPU cores on the Jetson TX2. Centering a camera right behind the windshield, in such a way that it is parallel with the line of sight of the user, the Jetson uses the machine learning api to draw boxes around objects it can detect (people, TV monitors, laptops, phones, etc..).

The data that is sent is a comma-separated string with the left, bottom, right, and top corner being sent in that order. Because multiple objects can be detected in a single frame we can send multiple bursts of this data before stopping. A stop message is sent after the objects are all transmitted to indicate the end of the data in a single frame.

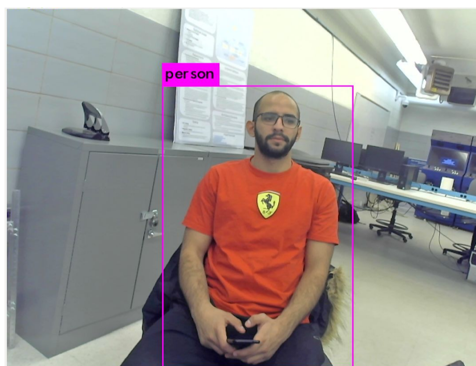

Figure 4: TinyYolo Object Detection

2.3. Graphical User Interface

The GUI in this case merely projects the coordinates of the objects in question. Our Jetson takes video feed from the camera at 416x416 resolution, whereas the the Jetson itself has a default resolution of 1920 x 1080. The coordinates of the GUI are offset to (1920 - 416) / 2 to accurately project the box around the object on the windshield. The GUI (or HUD) was implemented using Python for the code. The HUD reads data from a file which is in JSON format to break each object detected into a separate box and display them on the screen in their respectively shifted coordinates. This leads us to the final stage of implementation, connecting all these components together. This implementation is done by using an interpreter as a server for all of these clients to sync up their information.

2.4. Interpreter

The interpreter is the component that connects all of the sensors and the GUI together. It has two UDP sockets open that will receive data from the eye tracker and the object detection. When new data comes in it will adjust the data to fit the screen and then it will determine whether or not the user is looking at a given object. It does this by checking if the eye coordinates are within the coordinates of any given object. It will then write to a json file describing the location of the objects that are not looked at. The data that is sent from the interpreter and the GUI is via a JSON file in the following format: { “object #” : {“x”:x_position, “y”:y_position, “w”:width, “h”:height}}.

Below is a topological diagram of how the system works. The arrows show the direction data flows from each device to the other to achieve the said purpose.

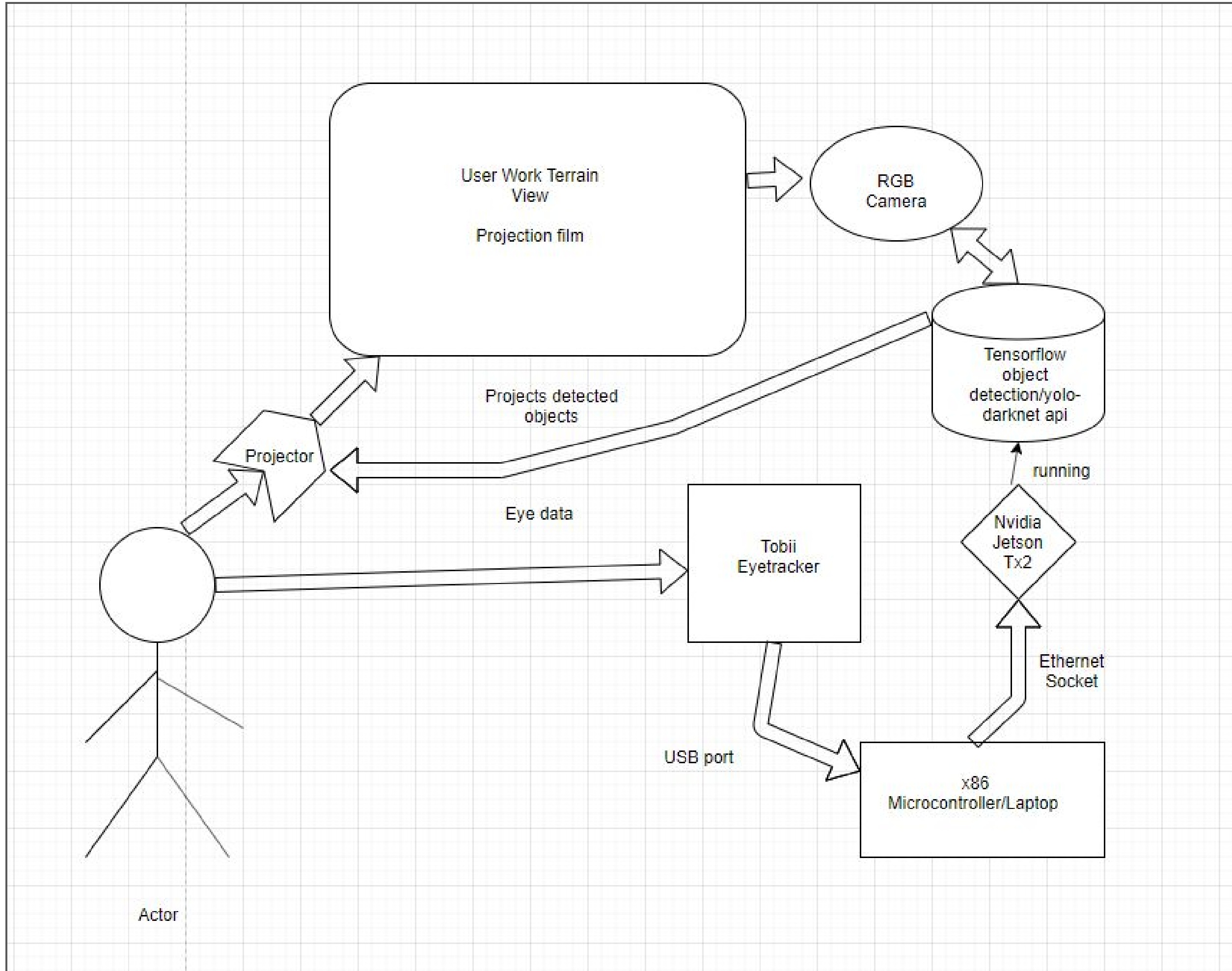

Figure 5: Hardware Design with Eye Tracker Connected to Jetson TX2

2.5. Limitations and Challenges

Due to our location, there were several limitations we were constrained by. Some were due to the tools we were given and others due to where we are on campus:

I. Tobii Drivers: The first constraint is that the Tobii would not connect to the Jetson directly via USB cable. We had to go around this issue using sockets and a third party tool as a server to manipulate data. In the current implementation the Tobii Eyetracker 4C is connected to a laptop and data is sent to the Jetson via ethernet using the laptop as an intermediary device.

II. Hardware: Where the Jetson TX2 is a top of the line module for parallel computation, object detection was still not as fast as it should be. Initially using YOLO api, the detection program would get about 3-5 fps when detecting an object. We would get around 11-13 fps when using the tiny version of YOLO (TinyYolo), which excluded a lot of information to run faster. Finally, the system used low level commands to increase GPU usage, which brought the frame rate up to an acceptable level 22-25 fps when detecting 1 object, falling below 10 when detecting more than 3. This is all done on a computer monitor. When including the projector and GUI, and detecting multiple objects we would get around the same result.

III. Latency: Due to the fact that the system uses intermediaries to communicate between the 2 main modules of our program, there is a lag in implementation. The team did their best to eliminate this latency to an acceptable threshold however given the appropriate drivers, this system may run much faster in real time.

IV. Location: Due to being on campus and utilizing the senior design lab as our main environment, we were not able to test this system in a moving vehicle. However, our client did not require this as this project serves a proof of concept which may be abstracted into other things.

3. Testing

Testing the system was no small task considering all the different components including both hardware and software. The team utilized a combination of unit, integration, and system tests.

Building our system involved a lot of unit testing at the start. The team wanted to ensure individual pieces of hardware and software performed to our specifications before trying to connect components together. For example, when initially working with object detecti on we didn't even plug in the eye tracker or projector. The team wanted to make sure object detection was fool proof on its own before moving on. Testing object detection was done by running it on the jetson with one external camera connected. We could very quickly be able to tell if it was working correctly or not because it would draw a box around the object and say what it is. The team repeated this process with the Tobii Eyetracker, testing that it worked solely on its own on a stand alone laptop. As object detection and eye tracking used existing software, the testing in this stage of the project was mainly ensuring it worked on the systems used in this implementation.

Once the team were confident the pieces of hardware could run the software correctly, we started integrating. An example of an integration test run was sending Tobii's eye coordinates to the Jetson. Since the Tobii was running on a different laptop we had to send the data to the main computer which is the Jetson. This is the first step of integrating our project. The team repeated this type of testing with sending data to the GUI. Testing everything wasn't necessary at this stage, so just being able to send data between hardware was a successful integration test.

Finally, system testing ensured the overall system worked together. Once it was known that the individual components worked, and could communicate with each other, it was time to test the entire system. For this step the system the team utilized manual testing to forge the way to the final solution. The system would run, then by having a human in the 'driver's seat', they would see what needed to change for the next iteration. A lot of this stage was adjusting the Tobii calibration or where the camera was pointing. This was also the shortest of the three because after unit and integration testing the system 85% done. It did take a substantial amount of time to integrate the system into a single cohesive unit.

The last major part of this stage was building the frame. Testing originally occurred by holding the windshield up manually. This obviously wasn't ideal because it was hard to repeat exactly where we had it before. Thanks to Danfoss, we were able to get some 8020 aluminum profiles that we put together to hold the frame up. Setting everything up was fairly straightforward

and after ensuring our projector was in the right position everything ran to the specifications laid out by the client, thus passing the system tests.

4. Related Work

Tobii, who produces the eye tracker used in this project, has their own work with eye tracking in the automotive field. Much like this project, Tobii seeks to watch the user's eyes and alert them when they are distracted, either drowsy or looking away from the road. Unlike our project, which uses a full size Tobii Eye Tracker 4C, Tobii's own automotive project uses the Tobii EyeChip. A pro of this approach is that the system can recognize different users and can interact with the car to change the seat position, mirrors, etc. which is beyond the scope of this project. The cons of this approach is that it interacts with the dashboard of newer cars, meaning without a smart display system in the car, the user cannot use the alerts that the Tobii system would send to them. Our approach seeks to fix this problem by adding a HUD which can be installed into any existing vehicle, allowing for a broader user base than for the Tobii automotive system as it is in its current state.

Nuance, the company who created Gaze which is a software that integrates with the eye tracker, has produced their own car which monitors user's eyes and displays alerts. In their implementation, a tablet that runs windows to display notifications to the user when applicable. The pros of this approach compared to ours is that it implements microphones that allows the system to look up information for the user and either display or speak it back using an assistant like Cortana on Windows. A con of this approach is that the tablet is mounted to the dash, meaning the user must look down to view the information whereas our implementation allows the user to continue to look at the road or environment in front of them as information is added to the HUD.

Appendix

Appendix I - Operation Manual

1. Required Equipment

- Jetson TX2 or another linux computer
- Tobii Eye Tracker 4C
- USB hub
- Windows 10 based device with ethernet and USB capabilities
- USB camera
- Short throw projector
- Screen with projection film applied
- USB Keyboard and Mouse

2. Hardware Setup

First setup screen with the projector sitting up against the screen. Then connect the Jeston with the projector, mouse, keyboard, and camera. Afterwards connect the windows device to the Jetson with an ethernet cable.

3. System Setup

3.1. Ethernet Setup

To setup the connection between the Jetson and the windows device follow the steps below:

1. Get IP address from windows machine
2. Open network manager on the Jetson
3. Add a new ethernet connection
4. In IPv4 settings
    a. Set to manual
    b. Add a new address with the same three numbers for the first three numbers and a different last segment.
    c. Set the subnet mask to 255.255.0.0
    d. Leave the gateway blank
5. Save the settings
6. Check connection by pinging the Jetson from the windows device. The ip address should be the same as the one set in the ethernet connection. On the Jetson, this is also listed under eth0 from ifconfig.

3.2. Projector Setup

Use the corner adjustment setting for the projector to adjust the image to fit correctly on the projection film.

3.3. Eye Tracker Setup

Download and install the drivers for the Tobii Eye Tracker 4C from Tobii's website. Then follow the instructions to calibrate the eye tracker (This can also be done additional times in the future for different systems).

4. Software Setup

4.1. Windows Software Setup

The software for the Windows system is all in the Eye_Tracker directory on the "Development" branch in the git repo. This needs to be pulled to be used.

4.2. Linux Software Setup

The code for all of the linux system is in the master branch of the git repo. Some of the code needs to be compiled before hand. To do that just go to the following directories and run "make".

- sddec19-14/linux_core/
- sddec19-14/life/darknet/

5. Running

1. Get IP address of the Jetson on Eth0, this should be the same as when it is set during ethernet setup.
2. Start GUI by running "python3 dhud.py" in "sddec19-14/linux_core/gui/"
3. Start interpreter by running "./startup" in "sddec19-14/linux_core/"
4. Start the TinyYolo by running "./startTiny.sh" in "sddec19-14/life/darknet/"
5. On the Windows system, start the Tobii code by running the "Interaction_Streams_101" from the "sddec19-14/Eye_Tracker/" folder on the "Development" branch of the repo.
    a. Enter the ip address of the Jetson from this command line
6. Wait for all components are connected, the interpreter should output some information about the camera and eye tracker being connected.
7. Bring up the GUI window to see it in action.
8. Press 'q' at any time to quit the GUI if needed.

6. Expanding on Implementation

The system is designed to be easily added to and taken away from. To add a different sensor simply create code to run and get data for the sensor and write to a socket and add a routine to read from the socket. To remove code just remove the routine from the interpreter. If any changes need

to be made to a sensor, just modify the code that runs that sensor and make sure the data format sent over the socket is the same as before.

Appendix II - Alternative Designs

1. Alternative Heads Up Display Designs

Over the course of this project there were several alternatives to the python program that ultimately became what is used to display information from the interpreter to the user. C/C++ was considered as a possible way to create the HUD at the beginning of the project. This was scrapped as support for basic visual application heavily relies on deprecated technology such as graphics.h.

The majority of the alternative designs for the HUD revolve around using JavaScript in various forms. The first inception of using JS for the front end was just a HTML page with JS embedded in it. This solved a lot of our needs as it was lightweight and could produce a full screen projection that was necessary for this project. The problem with this approach is that CORS (Cross-Origin Resource Sharing) blocked any attempt to read the data file from the Jetson. While this wasn't an issue during testing as the development environment allowed such calls to happen. The application used was Brackets which does not have support for the OS the Jetson was running and attempts to solve the CORS problem were fruitless.

The next iteration of the HUD attempted to solve the CORS issue by using Node.js. This initially seemed like it could be what would solve the problem because Node hosts the files on a local server. The problem with the Node approach is due to the server hosting static content while the system demanded dynamic content. This lead to the scrapping of this approach in the hopes of a better alternative.

The final approach attempted at creating a HUD with JS utilized React. React solved the CORS issue by importing the data file directly into the HUD file. This initially seemed to solve the issues as it would display and update data. The issue with React in its development environment is that each time the data file was updated, the entire program would recompile and refresh. This caused a substantial delay in its effectiveness and looked choppy. React-Hot-Loader alleviate the issue of reloading the page but did not help with the speed of the application. React requires all files to be under the 'src' folder which means that the data file must be in that folder, which triggered the application to recompile, causing about a second of delay. Trying to get the file from outside the src folder using the fetch api also had the same CORS issues as other JS implementations. The constant recompiling of the application was unacceptable and other solutions were explored.

After this the team looked into switching from using a file to using a socket to read and write data that the HUD. There were many 'almost solutions' but in the end React was not made to read and write data from a socket and that is when the HUD using Python was developed which is able to read and update data on the windshield at pace consistent with our requirements.

2. Alternatives to Using a Laptop as in Intermediary Device

Due to the lack of support for the Tobii Eyetracker on ARM systems the eye tracker requires an intermediary device to collect and send data to the Jetson. We decided to generate other solutions that were ultimately scrapped as well:

I. First solution was to change the eye tracker and use something like google glass. This suggestion was immediately shut down by the client as he wanted the system to not include any eyewear and use what was abundant.

II. We found an alternative to the Jetson which supported an x86 architecture and would allow us to hook the tobii eyetracker to it directly with no use of an intermediary device. This machine is called the Latte Panda. The Latte Panda we received had many parts missing, including a storage device such as an M.2 SSD, hence installing an OS on it proved impossible. We decided to switch back to the Jetson to keep costs down to a minimum.

III. Emulate using WINE or QEMU. The instruction set on the Jetson allowed it to support Virtual machines. We thought about including a virtual machine that can communicate with the Tobii eyetracker, but having this run in conjunction with the heavy GPU usage of the object detection API would render the project slow and useless.

3. Alternatives to Object Detection

Initially we wanted to perform Object detection manually using OpenCV. This included a process named template matching using OpenCV python scripts. This solution would not work in real time as there are lots of complex computations to be done. These computations can be done in parallel using a machine learning api that makes use of the cores on the Jetson.

Appendix III - Other Considerations

1. Jetson TX2 Development

The majority of this project revolved around getting the Jetson to play nicely with the other components we were given. Getting the Jetson working and communicating with the other components of the system was a bottleneck to development. Initially, even getting an operating system onto the Jetson proved to be an issue and required many attempts. The system setup now is not ideal. Further work on a project that seeks to do as this one would benefit from using an alternative to the Jetson. Development on this project could benefit from having all processes and external components connected to a single system that handles all processing for the system rather than communicating data between devices.

2. The Sunk Cost Fallacy

In the development of the HUD a majority of time was spent trying to make it work using the programming language of choice by Aaron. That language being JavaScript as that is what he uses for his job day to day, having many features that allow for lightweight development and

making the code very modular. If need data needs to be displayed to the user, an additional field can be added to the JSON objects passed to the code that produces the HUD. This led to many weeks of looking into solutions to the problem of reading a local file using a language primarily used in web development. There are many 'almost solutions' to the issue of reading a file from the local file system.

The final nail in the coffin for the JavaScript idea for the HUD came when looking into using a socket to communicate data. There are options for using sockets in JS on Firefox OS which was not an option for this project. After switching to Python, the HUD was up and running and is used in the final iteration of the project. While the code used in the JS implementation was the basis for the Python version of the HUD. This is all to say, one of the learning experiences over this project is learning to move on to a new alternative faster. While the research into communicating with the rest of the system was a learning experience into what the limitations of JS, the amount of time spent trying to fix the issue with small glimmers of hope that turned out to be dead ends could have been reduced.

The biggest issue is that in each scrapped iteration of the HUD, it almost worked as was needed for the project. This led to trying to get the last 20% of functionality working longer than should have in retrospect. Python bridged the gap that allowed the HUD to function at a level consistent with the functional and non-functional requirements of the system.