Robotic Teleoperation via Motion and Gaze Tracking

A Major Qualifying Project
submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfilment of the requirements for the degree of
Bachelor of Science
in
Robotics Engineering

by

Nicholas H. Hollander

Date
May 6, 2021

Report Submitted To:

Professor Zhi Li, Advisor

Abstract

Robotic systems have traditionally been controlled by handheld interfaces such as game controllers, joysticks, mice, and keyboards. While these interfaces are highly effective, there are situations where an operator may be unable to use their hands. The proposed method explores the use of an Intel RealSense depth sensing camera as a robotic teleoperation input method by tracking the operator's eye-gaze to the computer screen. It utilizes the eye-gaze vector and body position to determine operator intent and provide input to a robotic control system.

Acknowledgements

The success of this project could not have been achieved without the assistance of many exceptional individuals in the WPI community. These people provided knowledge, direction, and direct assistance that was valuable to the completion of this project.

Special thanks to Karim A. Tarabein and Aashirwad C. Patel whose work in RBE 526 (Human-Robot Interaction) was instrumental to the success of this project. Karin and Aashirwad both provided critical assistance with research, code, and algorithm development.

I also want to express my sincere gratitude and appreciation to Professor Zi Li of WPI's Human Inspired Robotics Laboratory for serving as the primary advisor on this project and providing important guidance and input on HRI design that proved essential to the completion of this project.

Contents

List of Figures

List of Tables

Executive Summary

Throughout the many fields which make use of robotic technologies to control manufacturing, medical, or other processes, there are times where manual human control is favored over automation and autonomous robotic control. In these situations, the operator of the robot typically utilizes a keyboard and mouse, joystick, or game controller to provide input to the robot. Unfortunately, these methods have a limited number of real-time inputs, forcing operators to switch tasks in order to adjust cameras, switch manipulators, or perform other high-level tasks. In order to facilitate effective operation of a robotic system, the cognitive, physical, and temporal workload on the operator must be minimized. The use of head position and eye-gaze vector tracking allows for an intuitive secondary control input stream to be provided to applications, minimizing cognitive overhead caused by task switching for high level control operations.

Outside of industrial robotic control, there are other situations where the use of visual tracking for head and eye-gaze vectors can be extremely useful. Many people suffer from disabilities which limit or completely remove their ability to operate a controller with their hands. Under such circumstances, the only methods of robotic control are those based on body and eye position interpretations.

There are commercial systems available today, such as the Tobii Pro line of eye tracking devices[1], which are research-enabled versions of their consumer grade Tobii line of eye tracking products[2]. While these devices are extremely capable, they require expensive specialized hardware, and the consumer version is restricted for use in non-research applications. At this time, there are very few other commercial eye tracking solutions available on the market.

The goal of this project is to tackle the lack of low-cost and versatile eye tracking systems by implementing a high-fidelity body and eye-gaze tracking system utilizing the low-cost Intel RealSense family of RGB+D cameras. These cameras provide a high resolution video stream with an associated depth stream. The project aims to digest the RGB+D data into streams of data which can be easily implemented into other projects via ROS or by importing a Python library.

1 Introduction

1.1 Motivation

Most human-operated robotic systems are controlled using handheld interfaces such as a keyboard and mouse, joystick, or game controller. These forms of input are time tested and highly reliable, but there are many situations where such a control system may be less appropriate, or altogether impossible to use. This project aims to develop a form of control input based on an operator's head position and eye-gaze vector.

1.1.1 As an Additional Input

While preoccupied by certain tasks such as performing surgery, driving a car, or controlling a robot with a game controller, an operator may be unable to dedicate their hands to an additional task. In these circumstances it may be possible for the operator to digitally or physically switch the input to control another task. However the need to switch inputs in this manner increases the cognitive workload on the operator, resulting in a corresponding increase in the likelihood of making an error.

Utilizing the head and eye position to provide an additional source of inputs allows the operator to control other aspects of the system without compromising their original control, and without dramatically increasing the cognitive workload. This additional control channel can be assigned to camera control tasks, or to direct high level activities for the robot, such as visually selecting a target.

1.1.2 Disability Service

A number of medical conditions may result in an operator losing the ability to use their hands. With most modern robotic control systems built around the user's hands to control the primary inputs, this can result in significant barriers for the physically disabled. A variety of medical assistance telenursing robots are currently being developed in academic and research spaces. However, much of this research is directed at developing assistive robots for However, many of these robots require an able-bodied operator to provide inputs. An effective head and eye-gaze tracking system could enable a disabled operator to direct a robot through visual queues.

Recent research in the subject of disability service robotics can generally be broken down into two major categories. The first category is nursing staff assistance systems. These are systems such as the Tele-Robotic Intelligent Nursing Assistant (TRINA)[3] which expand the physical capabilities of nurses, doctors, physicians assistants, and other medical professionals to effectively care for patients without being physically present. With few exceptions, these robots require the robot operator to control the robot via the aforementioned physical contact interfaces. The second category is patient assistance systems. These systems are designed to interact with and operate under the direct command of a physically or mentally incapacitated patient. These systems are not as sophisticated as existing telenursing platforms, although promising research is being done with patient-friendly interface methods, with speech-to-text[4] receiving the lions share of development.

1.2 Current Technology

While head and eye-gaze tracking technology does not currently enjoy widespread adoption in the field of robotics, there has been a significant amount of research performed on this topic and related subject areas. There are a variety of eye-gaze tracking systems available today that cover a wide range of applications and deploy a number of different technologies[5]. Since 2010, with the introduction of the Microsoft Kinect as an inexpensive research tool, there has been widespread availability of 3D body tracking systems, which also provide a potential high-fidelity alternative input channel for robotic systems.

1.2.1 Gaze-to-Screen Tracking

Some of the earliest digital eye tracking systems relied on corneal reflections from infrared light sources shined into the eyes of the operator[6]. This technique, also known as Purkinje imaging, calculates the relative angle of the eye by measuring the angle between the center of the eye and the reflected dot[7].

Later systems made use of increased RGB camera quality, and/or configurations utilizing multiple cameras to capture data with increased quality. A large number of models have been proposed that make use of both single and multi-camera setups[8].

In addition to the Purkinje system for tracking eyes, a number of other methods have been developed utilizing different forms of tracking technology. The most common view-gaze detection approaches, hardware requirements, and processing methods are summarized in Table 1.1.

Gaze Detection	Hardware	Analysis Method
Pupil Tracking	RGB, Near IR	Analysis of image contrast
Corneal Reflection	RGB with LEDs on 4 corners of computer screen	Analysis of position of LED reflections
Multi-Camera Shape Matching	Stereo camera (RGB/Near IR)	Analysis of eye and pupil relative to 3D template to establish gaze vector

Table 1.1: Primary Methods for Gaze Tracking from Kar et al

Pupil-Tracking cameras represent the most common and basic systems currently available for gaze detection. They require little to no specialized hardware, and can be easily developed using low-cost consumer and professional grade imaging devices. These systems use computer vision to locate the the operators pupil, and derive gaze-vector information based on its relative position to other eye and facial features. Corneal Reflection systems, also known as Purkinje imaging, takes advantage of the reflectivity of the various layers of the optical structures within the eye. In order for this system to work, high quality and high sensitivity cameras must be used, as the reflected light patterns are difficult to capture without the use of specialized hardware. The final approach is multi-camera shape matching. This approach captures multiple images of the eyeball from a variety of vantage points, and utilizes feature matching to determine the position and rotation of the eye in 3D space. This approach makes use of the technology used in the other two detection methods, and produces extremely high quality data. Like Purkinje imaging, this approach requires relatively expensive and high-quality image sensors. [7][8]

With the widespread availability of consumer grade RGB+D depth sensing cameras starting in late 2010 with the introduction of the Microsoft Kinect for the Xbox 360, various algorithms and approaches were developed to take advantage of the new dimension of data available[9][10][11][12][13]. The approaches and methods introduced by these research papers provided a large portion of the foundational knowledge from which this project was constructed.

1.2.2 Body Motion Tracking

The introduction of affordable depth tracking cameras also allowed for a significant amount of development and research to take place in body tracking systems. Previous systems had either severely limited accuracy (2D tracking), or cost thousands of dollars (Vicon 3D system). Later developments in Inertial Measurement based tracking systems allowed for a person to be located

in 3D space by deriving component translations from the acceleration, rotation, and gyroscopic transformations. The most common body tracking methods are described in Table 1.2[14].

Capture Type	**Hardware**	**Method**
Optical (3D)	Single Camera RGB Multi Camera RGB Single Camera RGB+D	Markerless or Passive (Retroreflective) Markers
Optical (2D)	Single Camera RGB	Markerless
Inertial	Accelerometer, Gyroscope, and/or Magnetometer	Mathematical Position Derivation

Table 1.2: Methods for Motion Tracking from Yayha et al

Like the aforementioned gaze tracking systems, position tracking systems can generally be broken down into three categories. The first category is 3D optical tracking systems, which utilize one or more optical cameras to determine the position of a body in 3D space. Newer tracking setups may also make use of depth-sensing cameras to provide an additional data channel. These systems rely on computer vision processing of the RGB and/or depth streams or retro-reflective dots (As see in in the VICON) system to locate the body. The second category is 2D optical systems. These systems typically rely on a single RGB camera and computer vision feature matching algorithms to determine the screen space location of a target, or the relative angular position. Because of their limited vantage point, these systems generally do not have 3D location capabilities. The final common capture system is inertial, which relies on an accelerometer, gyroscope, and/or magnetometer placed at each joint on the body. By combining the derivation of these sensor readings, the accelerative and rotational components of an operators movements can be translated with a great degree of accuracy into absolute positional information. This system has several advantages over optical ones, since it does not require any external sensing equipment, and can not be occluded by passing objects or the body of the operator. [9][10][11][12][13][14].

1.3 System Design Overview

In order to facilitate the effective design of an eye-gaze and head tracking system, the application needs to be broken down into a collection of semi-autonomous components. The major components of the system are outlined in Figure 1.1.

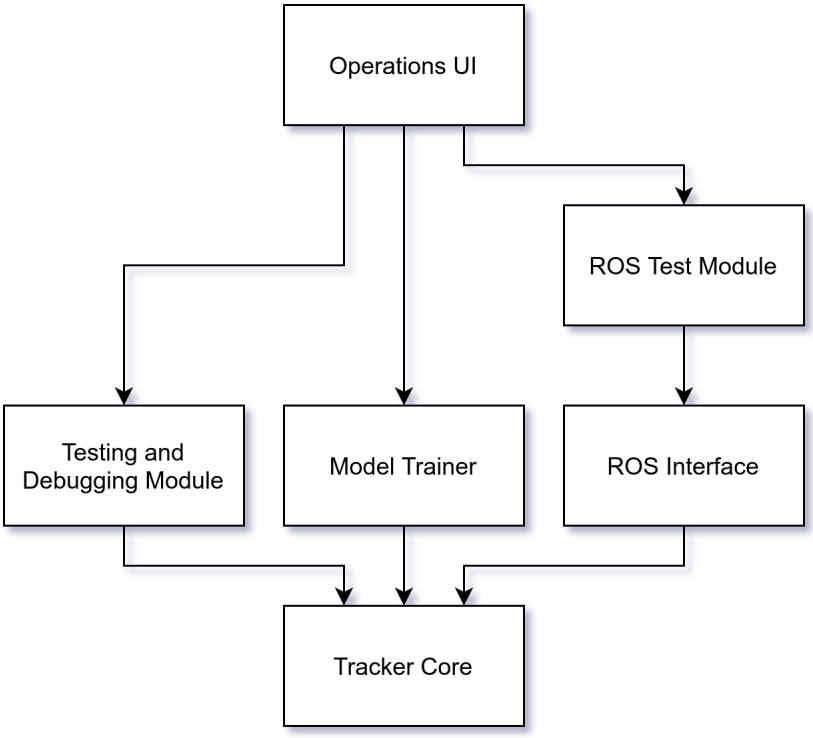

Figure 1.1: System Overview

1.3.1 Operations UI

The operations UI is a high level interface designed to provide easy access to the Testing and Debugging Module, the Model Trainer Module, and the ROS Test Module. This is a basic module which allows a user to interact with these three important debugging and development tools without direct interaction with the command line, which was a specified requirement for this project.

1.3.2 ROS Test Module

The ROS Test Module, along with the ROS Interface module, make up the bulk of the provided ROS code. When started, it will connect to the ROS interface and display the raw values captured by the tracking system.

1.3.3 Testing and Debugging Module

The Testing and Debugging module is primarily used during the development process for the Tracker Core module. This module does not have any specifically defined functionality, although it does interact with all of the exposed interfaces for the Tracker Core module. Development of the

eye-gaze and head position tracking system is generally performed between this module and the Tracker Core module before additional features are implemented in the other system modules.

1.3.4 Model Trainer

In order to provide enhanced accuracy for the tracking system, a Neural Network is used to relate extracted face and head position data and screen view coordinates. The Model Trainer application allows for model training data to be collected, and a Neural Network model to be trained. Each generated model is reliant on the size of the screen used, and the position of the RealSense camera relative to that screen, so each unique setup will require a differently trained network.

1.3.5 ROS Interface

The ROS Interface module is a sample implementation of the Tracker Core Python Library. It is a very lightweight module, mainly designed to provide a direct interface between the endpoints exposed from the Tracker Core module and other nodes which exist in a ROS cluster. If you are looking for an example on how to implement the Tracker Core Python Library in another application, such as embedding it directly into another ROS node or non-ROS program, this is a good Place to start.

1.3.6 Tracker Core

The Tracker Core is the meat and potatoes of the eye-gaze and head position tracking project. Internally, it contains several layers of logic which are responsible for retrieving the raw streams of data from the RealSense RGB+D camera, transforming the imaging data into a usable format, and then passing it through a stack of algorithms which convert the images into positioning information. The stack relies heavily on the OpenCV computer vision library, the TensorFlow machine learning library, and the advanced data manipulation features provided by Python.

2 Technology

The core module described in the Introduction of this document represents the application component responsible for performing the bulk of technical operations required to extract relevant derived positional information from a system operator. The core is implemented as a Python module which performs a series of sequential operations in a continuous loop. An implementing application only needs to call `init()` to configure the library, and then `loop_once()` for each frame to be processed. The logical flow of the application is illustrated in Figure 2.1.

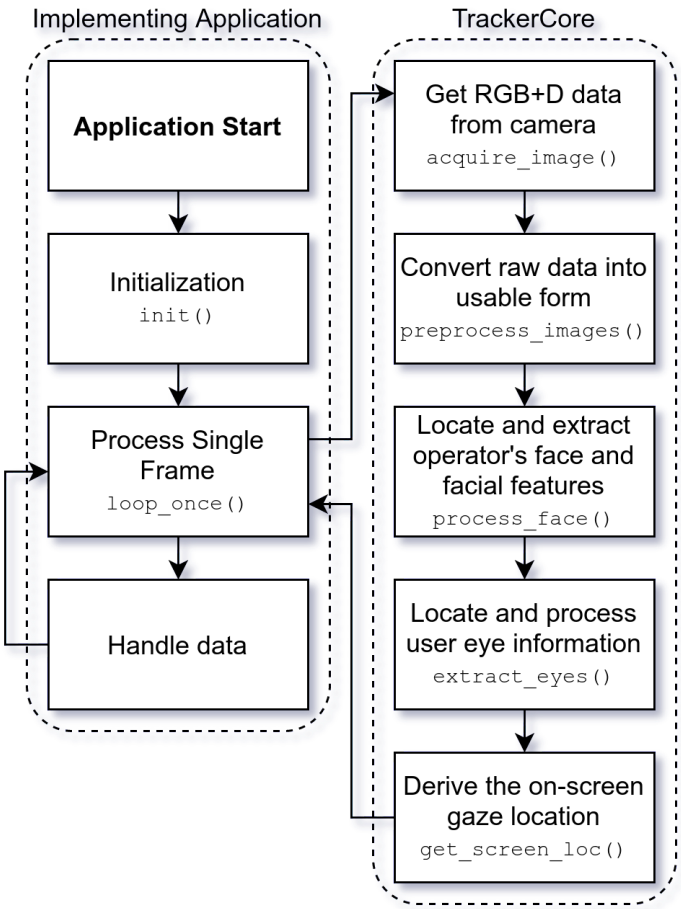

Figure 2.1: TrackerCore application overview

2.1 Logical Structure

2.1.1 Initialization

Found at `trackercore.py:init()`

The first step in the core processing flow is the initialization of the tracking systems dependent components, seen in Figure 2.2. When the implementing program calls `init()`, the core begins the configuration and subinitialization processes for the dependent libraries, as specified in the application configuration. This method is only called one time when the system is launched.

Figure 2.2: TrackerCore `init()` flow

2.1.2 Loop Controller

Found at `trackercore.py:loop_once()`

The loop controller is the second exposed method for implementing applications. When called, it will attempt to perform a single iteration of the acquisition, processing, and reporting flow. Assuming this process is successful, the desired data will be returned. If any step of the process fails or is unable to continue, the loop will be broken and no data is returned. This action begins by resetting variables and data structures in preparation for the next run. It then iteratively calls each of the sub-actions, until either the last one completes, or the process is aborted by one of the sub-actions raising an exception.

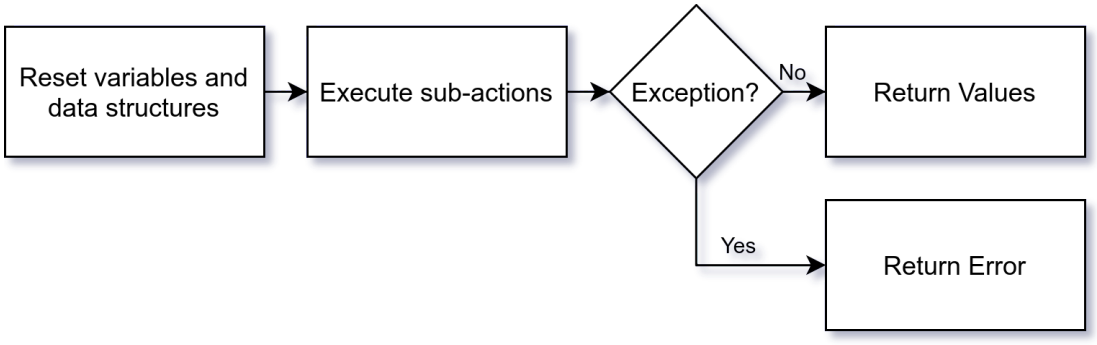

Figure 2.3: Trackercore `loop_once()` flow

2.1.3 Image acquisition

Found at `trackercore.py:acquire_image()`.

The image acquisition step is the first sub action called by `loop_once()`. When invoked, the acquisition pipeline attempts to get a color (RGB) frame, and a depth frame from the RealSense camera. Due to the internal logic used in the RealSense support library, it is possible that the depth frame may not be present, especially when the camera is first initialized. Assuming the data is present, the frames are saved to internal memory, and the process continues.

In the default configuration, the color frames are read at a resolution of 1920x1080, and the depth frames are read at a resolution of 1280x720, the maximum resolutions supported by this model of RealSense camera. If the tracker needs to be run on a lower-power platform, it may be possible to decrease these resolutions at the cost of tracking accuracy.

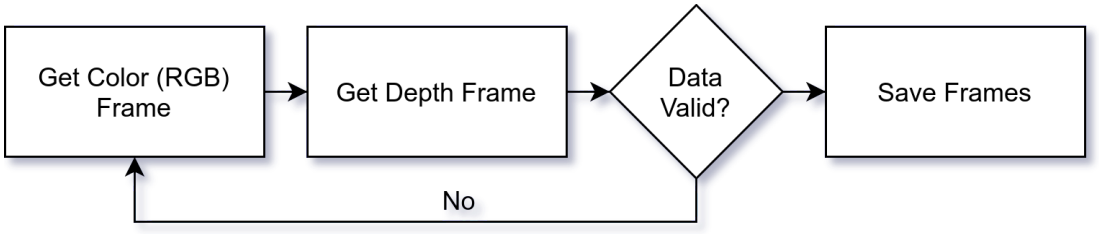

Figure 2.4: TrackerCore `acquire_image()` flow

2.1.4 Image Preprocessing

Found at `trackercore.py:proprocess_images()`.

Images returned from the camera are not in an immediately usable form. The separate RGB and depth sensing modules in the camera do not have the same alignment, and must be adjusted and converted into OpenCV compatible formats before they can be processed by the processing, extraction, and estimation algorithms. A colorized depth frame is also generated to assist in visualization and debugging purposes. The alignment and cropping process utilizes percentage based cropping parameters, which allows the function to operate regardless of the configured resolution from the image acquisition action.

Figure 2.5: TrackerCore `preprocess_images()` flow

2.1.5 Facial Processing

Found at `trackercore.py:process_face()`.

Facial Processing is the first major step in the processing pipeline. Applying the facial detection algorithm to a Full HD (1920x1080) frame requires an excessive amount of computing power, even on a relatively powerful machine. In order to reduce this load, the input image is downscaled during the grayscale conversion phase. A downscale of 50% results in a 300% increase in performance, without any noticeable decrease in the face tracking accuracy. This implementation makes use of `dlib`'s built in face detection algorithm and provided 68 point facial feature detection model, demonstrated in Figure 2.7, which provide a generally acceptable degree of accuracy, although these algorithms are comparatively sensitive to ambient lighting conditions, and have been known to suffer from accuracy issues on people with dark skin.

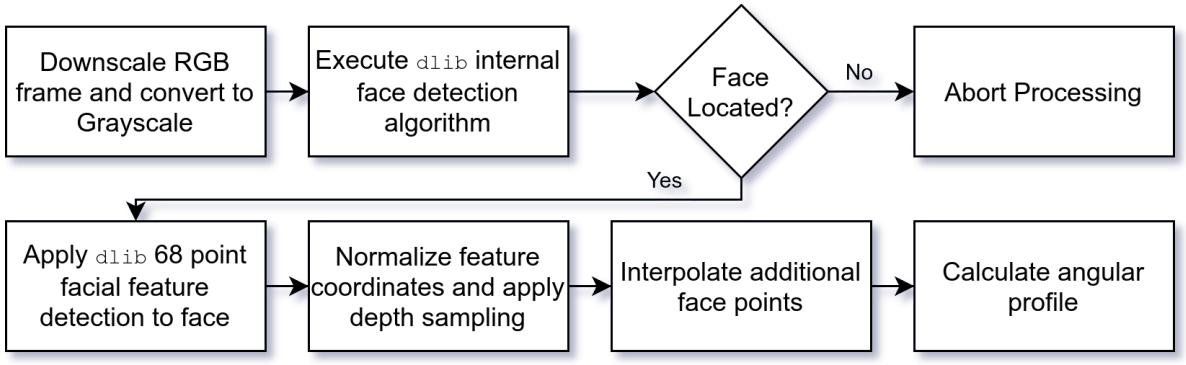

Figure 2.6: TrackerCore `process_face()` flow

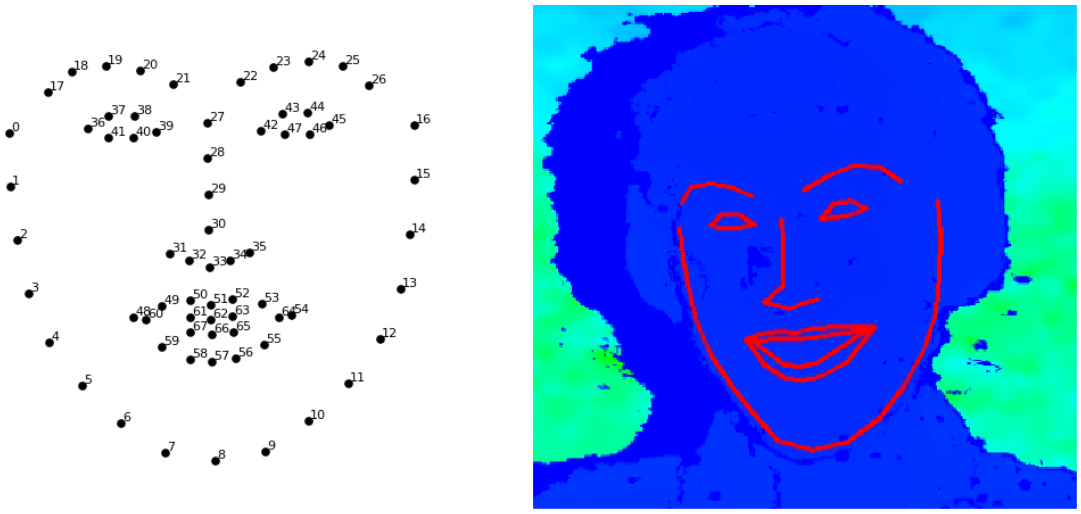

Figure 2.7: 68 Point facial feature detection applied over colorized depth frame

Feature Interpolation

While the 68 point facial tracking model does provide a great degree of accuracy, it is missing several points which are critical to effectively tracking the operator's head. The additional points as shown in Figure 2.8 and include one point at the glabella[1], one point on each cheekbone, and a point on the chin. The feature interpolation algorithm flow is outlined in Figure 2.9.

Figure 2.8: Enhanced feature tracking overlay with 4 derived points

[1]The glabella is the point on the face which lies on the forehead at the center of the brow.

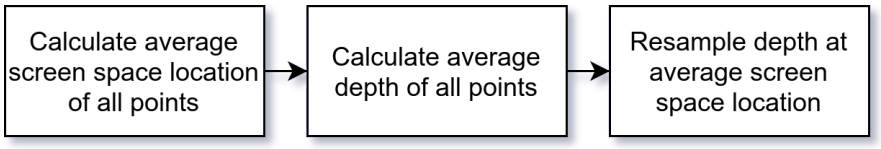

Figure 2.9: Facial point interpolation `derive_avg()` flow

2.1.6 Eye Extraction

Found at `trackercore.py:extract_eyes()`.

Once the operator's face has been located and features have been extracted, the eye extraction algorithm is run to determine the relative pupil locations. The extraction process outlined in Figure 2.10 is applied two times, one for each eye.

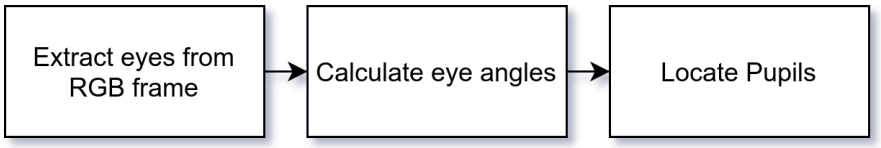

Figure 2.10: TrackerCore `extract_eyes()` flow

Extraction

The RGB eye extraction phase is carried out in two phases. In the first phase, the bounds of the eye are calculated using the normalized coordinates generated by `process_face()`, and those coordinates are clamped to $0..1$ in order to eliminate processing exceptions if the operator approaches the edge of the camera's view-frame. Once the bounds of the eye have been calculated, that subsection of the image is extracted. In the second phase, the 6 points that make up the boundary of the eye need to be transformed from the original image's coordinate space to the normalized coordinate space of the cropped eye image.

Eye Angle Calculation

After the points have been transformed, the corner points of the eye are used to calculate the eyes angle. This important piece of information allows the estimation phase of the algorithm to compensate for rotation of the operator's head.

Pupil Location

The final phase of the facial feature tracking workflow is the pupil estimation stage. This stage took the most time to perfect during the development of this project, eating up nearly 4 entire months of time, during which numerous solutions were attempted with limited to no success. A variety of convolutional classifiers were utilized in an attempt to automatically locate the pupils, and while the background research on the subject was extremely promising, limited knowledge in the field prevented these approaches from being successfully implemented. The approach utilized in this paper is a basic positional calculator which demonstrates sufficiently accurate tracking when used on low resolution data sets, such as those generated by the RealSense camera's color sensor. This process is outlined in Figure 2.11

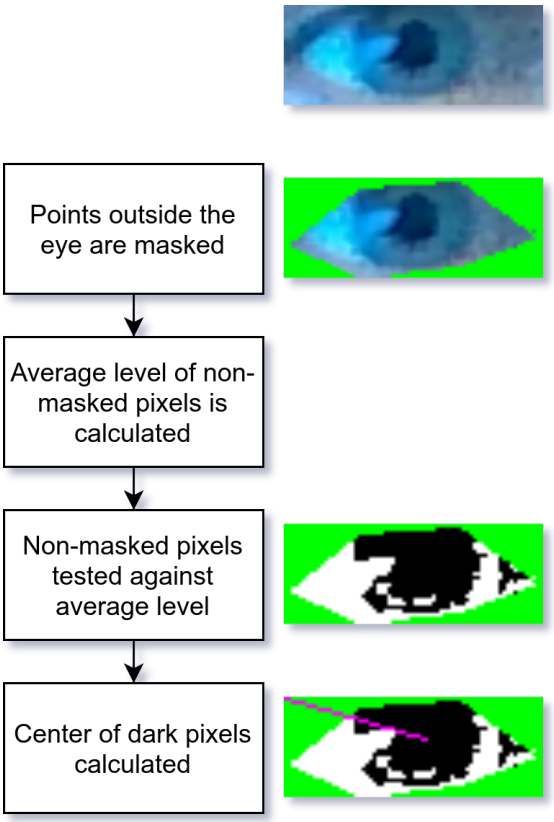

Figure 2.11: Pupil Extraction Algorithm

2.1.7 Screen Location Estimation

Found at `trackercore.py:estimate_screen_pos()`.

The final stage in this gaze-vector tracking saga is the screen location estimation system. This component aggregates a collection of essential variables from the previous stages of the loop, and

passes them through a basic neural network implemented with TensorFlow. This is the most basic step of the process, requiring a single call to `model.predict()` in order to predict the next set of values. When tested in semi-ideal conditions, with a single operator, the system had an average error factor of 10% to 20%, with significant accuracy variations depending on the location the user was looking at. Figure 2.12 shows how the model was far more accurate when predicting operator gaze at locations towards the center of the display (indicated in blue), while locations at the edges produced unusably inaccurate data (indicated in yellow and red).

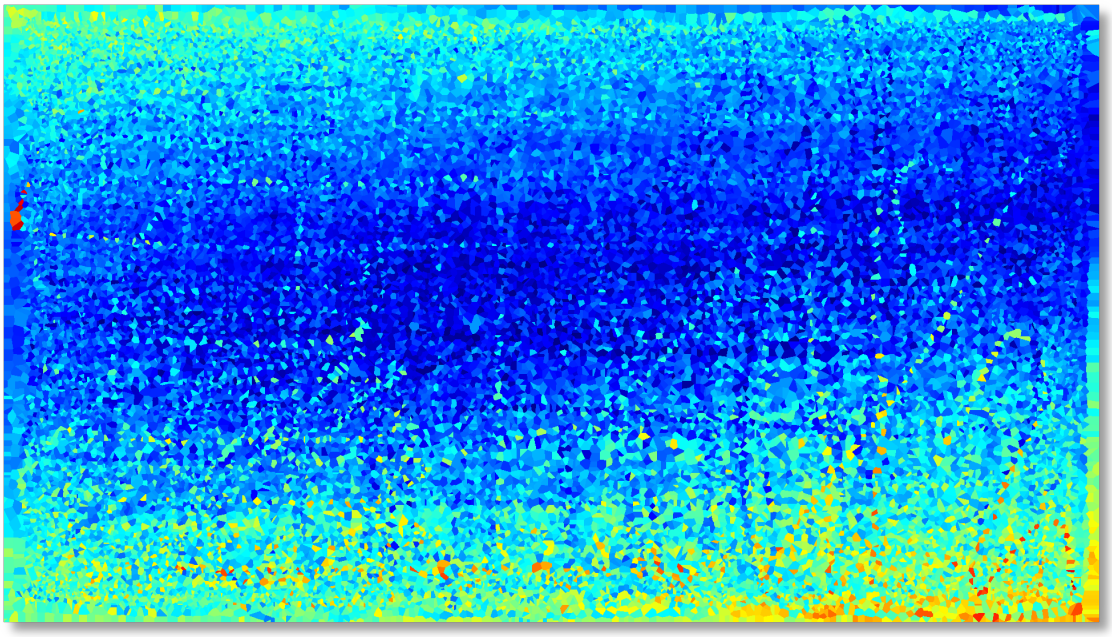

Figure 2.12: Model estimation accuracy by distance from actual position

2.2 Model Training

Because the tracker relies on a machine learning model, its effectiveness is highly dependent on variables such as the position and angle of the RealSense camera module, the size of the computer monitor used, and the quality and diversity of training data it is given. The data collection and model generation code is broken into two application components, with the data collector found in `trainer/collector.py` and the model training and generation code found in `trainer/train2.py`. These two modules an be accessed and controlled from the operations UI, however in order to achieve best performance, some tweaking may need to be made directly to these files.

2.2.1 Collecting Data

Collecting data is performed by launching the collector program, either from the terminal by executing `trainer/collector.py` or by selecting the ”Data Collector” option from the Operations UI. The data collector will launch fullscreen in a paused state as shown in Figure 2.13, and display the current readings it is getting from the Tracker Core. Pressing space will unpause the program, and begin recording training values. To ensure the highest possible accuracy, make sure that the test subject is staring directly at the mouse pointer for the entire duration of the test. If the user loses focus on the pointer, they can interrupt the process and delete the last few samples to prevent corruption of the data. It is recommended that between 20k and 40k samples be collected for each participant.

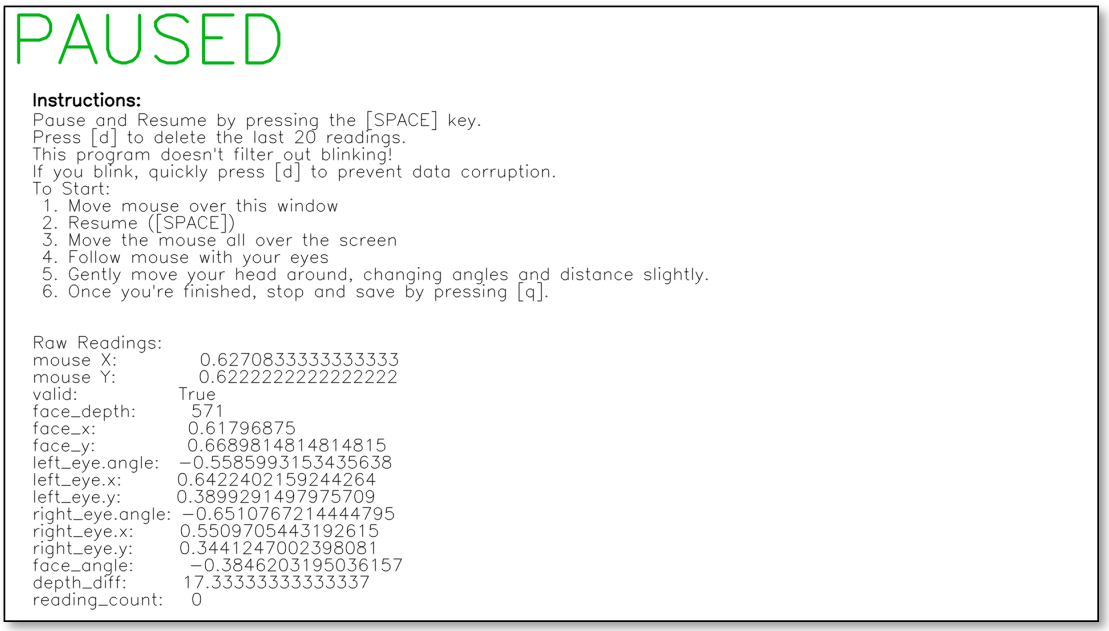

Figure 2.13: View of the training data collector when paused

2.2.2 Training the Model

Once a sufficient amount of data has been collected, the automated training process can be invoked either by executing `trainer/train2.py` or by selecting the ”Model Trainer” option from the Operations UI. This process can take anywhere from one to 48 hours depending on the computer being used, and the training parameters.

2.3 ROS

In order to facilitate the integration with existing robotic systems, an implementation of the tracker core as a ROS node was created. This node implements the basic functionality of the tracker core, and exposes it through several ROS topics and basic messages. A table outlining these messages is outlined in A.

2.3.1 Test Node

Found at `ros/src/eye_head_tracker/src/test_node2.py`.

This ROS node provides a real-time readout and overview of the tracker values, as well as an interface for adjusting configuration values. This node subscribes to all of the topics published by the interface node, and publishes to the command channel for sending configuration update messages. The test node also provides a readout of informational messages generated by the TrackerCore, which can assist in debugging.

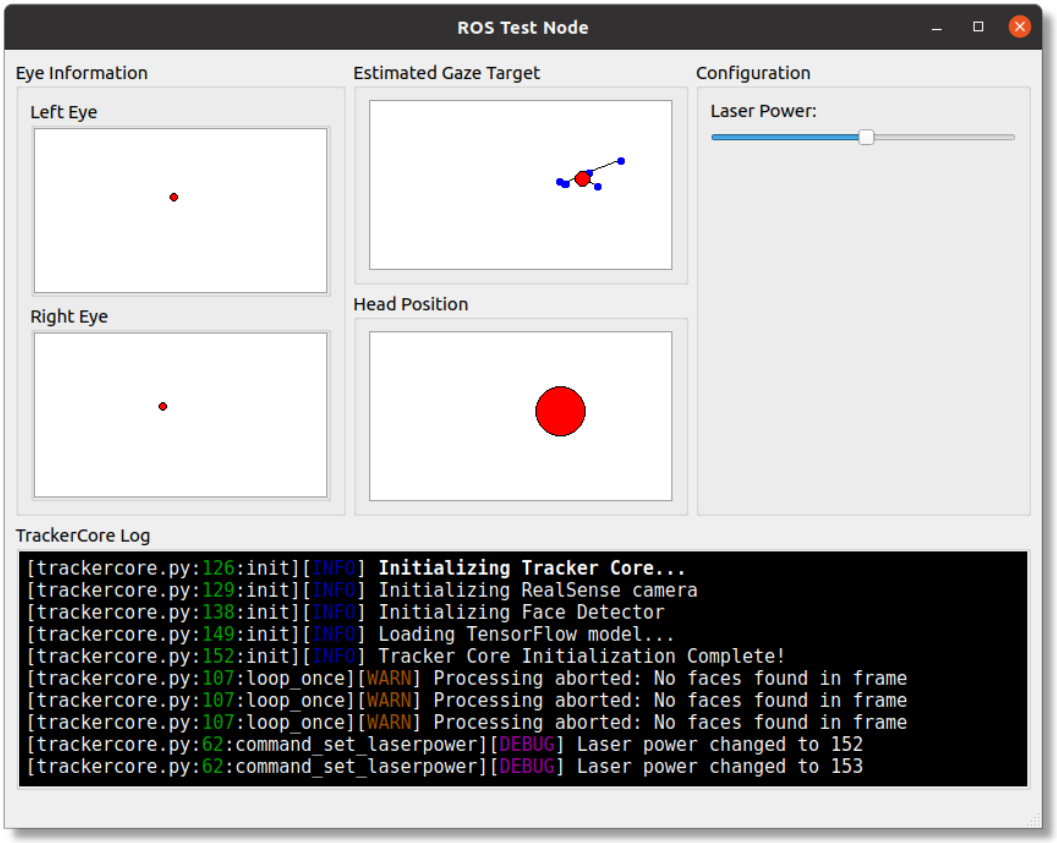

Figure 2.14: View of the ROS Test Interface

2.3.2 Interface

Found at `ros/src/eye_head_tracker/src/tracker.py`.

This ROS node provides a minimum working implementation of the tracker node over the ROS network. It subscribes and/or publishes to all of the nodes listed in Appendix A, as showin Figure 2.15.

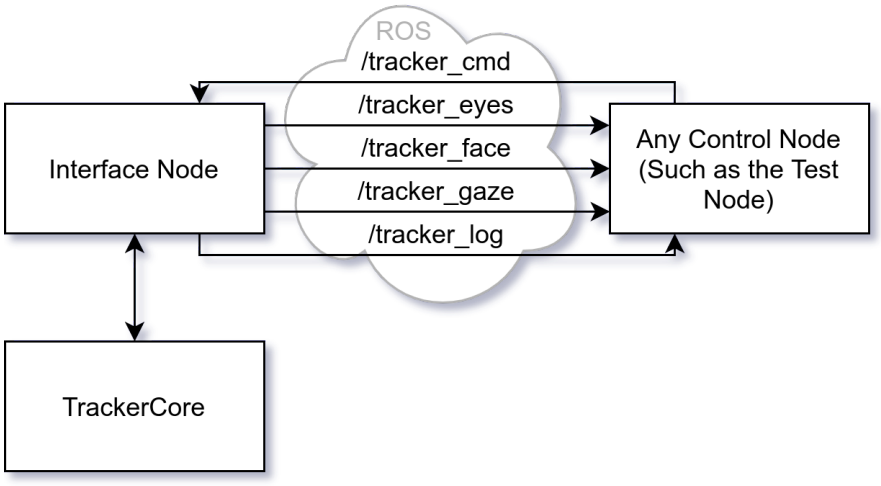

Figure 2.15: ROS Interface Diagram

2.4 Operations UI

Found at `userinterface.py`.

This module provides a user-friendly interface to assist in launching different components of the gaze tracking system. It contains a set of buttons, each of which will directly launch a subcomponent of the tracker system. For convenience and debugging purposes, `stdout` and `stderr` from each of these processes will be written to a terminal viewer. The interface has the ability to launch five sub-tasks, outlined below.

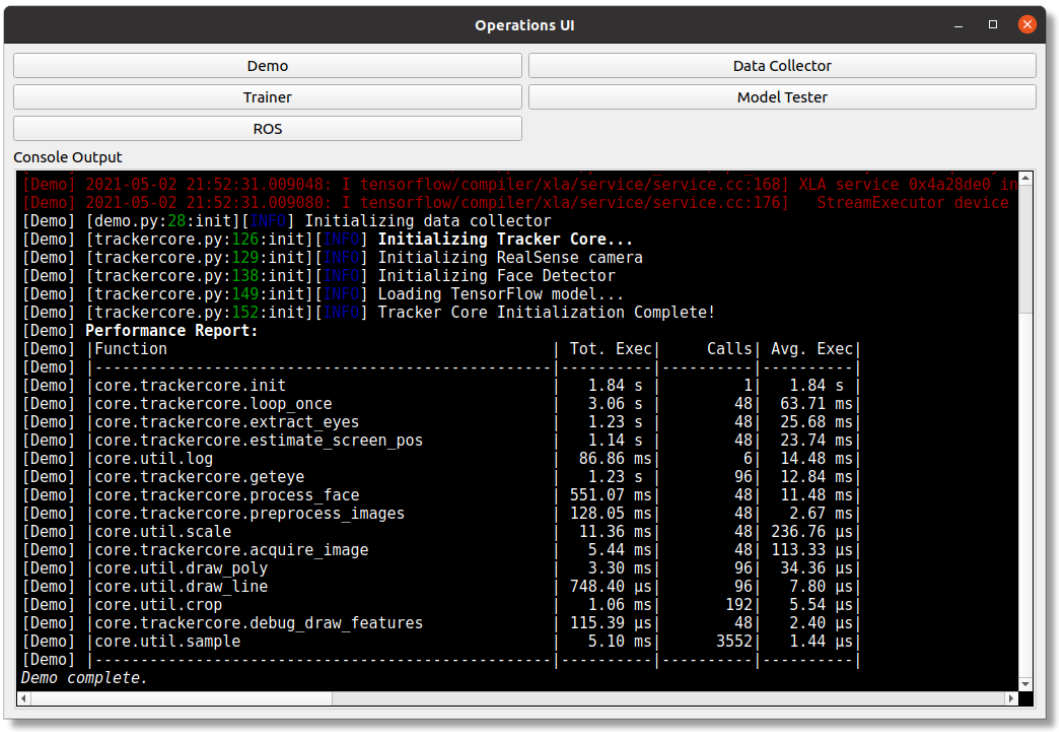

Figure 2.16: View of the Operations UI

2.4.1 Demo

Launched with `./trainer/demo.py`.

This button launches the demonstration interface. In order for this to work, you must already have a trained model in your workspace. The demonstration opens a full-screen window, and draws a series of points representing the last ten readings and their average, as shown in Figure 2.17. The black border on the image was added in post for clarity.

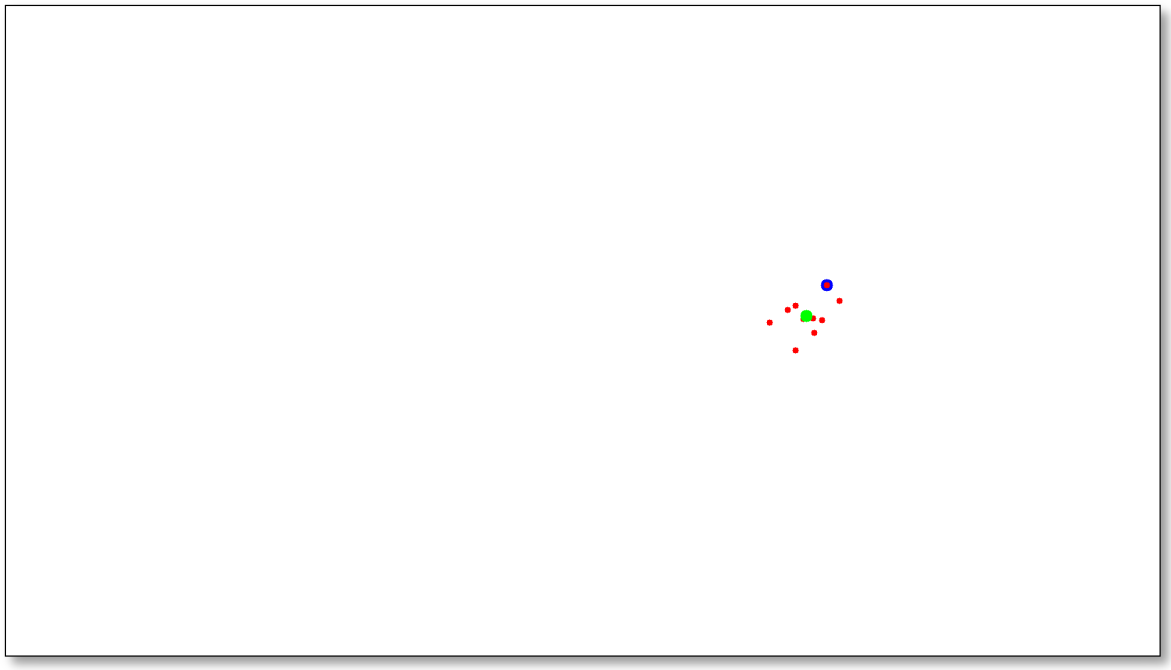

Figure 2.17: Demonstration UI

2.4.2 Data Collector

Launched with `./trainer/collector.py`.

This button launches the data collection module. The results of this module are written to `data.csv` for use by the Training Module. More information about this module is available in Section 2.2.1.

2.4.3 Trainer

Launched with `./trainer/train2.py`.

This button launches the Model Trainer using default settings. The model will be trained from the data written to `data.csv`. When debugging or fine-tuning the model, it is recommended that the training script be invoked directly from the command line. More information about this module is available in Section 2.2.2.

2.4.4 Model Tester

Launched with `./trainer/test.py`.

The model tester re-runs the training data through the model to determine how suitable the model is for eye tracking operations. The program will generate an average error factor reading, as

well as a graphic heat-map similar to the one in Figure 2.12.

2.4.5 ROS

Launched with `roslaunch ./ros/src/eye_head_tracker/demo.launch`.

Invokes the ROS launch file to start a ROS master, the interface node, and the test node. This spawns the interface seen in Figure 2.15. In order to use this button, your environment must be pre-configured with the ROS sources, which can be done by adding them to your `.bashrc` file.

3 Conclusion and Recommendations

3.1 Summary

This project explored the use of an Intel RealSense Depth Camera 435 as a low-cost replacement for more expensive eye tracking systems such as those manufactured by Tobii Pro and other research equipment manufacturers. Ultimately this project was not successful this year in producing a drop-in replacement system, however hope is not lost. There is a lot of room for improvement which can bring this system closer in line with the features and capabilities of these more expensive devices.

This section of the paper will go over the various areas of general concern in this project, and offer a number of potential remedies that can be implemented by future student groups who accept the challenge of pursuing this task.

3.2 Conclusions

3.2.1 Performance

In order to facilitate development and elimination of bottlenecks in the processing pipeline, a timing and debugging module was integrated into the core that allows for detailed post-run analysis fo the applications timing characteristics. These reports are written out to the console upon program termination, as well as to a file located at `/tmp/funcperf.csv`. A sample report can be found in Appendix B. This report will have more or less lines depending on what program was run, and what other functions were linked to the performance monitoring module.

Any function that relies on the `core` can take advantage of this performance monitoring by including `debug` and then placing the `@debug.funcperf` above the function.

The TrackerCore and associated programs were written in Python, which is a language known more for its ease of use and broad functionality than its speed. In the case of this application, that results in execution speeds of 20FPS, with the majority of time being split between `core.trackercore.extract_eyes` and `core.trackercore.estimate_screen_pos`, as seen in Figure 3.1. The eye extraction function relies on a nested looping analysis function, which incurs a severe performance penalty under python, and is responsible for the majority of the

26ms that is spent in that function. The other major time sink is the gaze estimation function, which involves invoking a TensorFlow model, which is an unsurprisingly slow operation. Both of these functions could likely have their performance improved through the use of clever mathematical tricks or re-implementation in a different language.

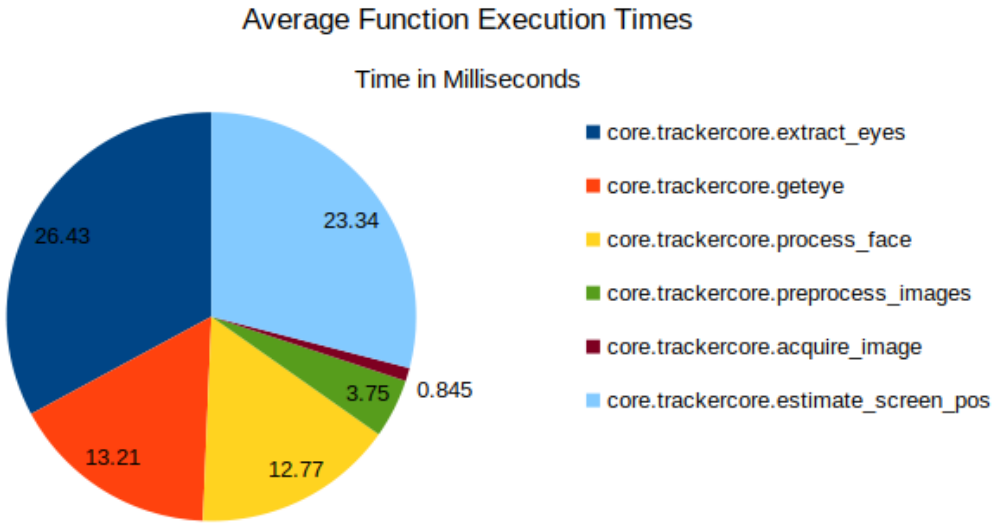

Figure 3.1: Function Runtime Performance Analysis

3.2.2 Accuracy

Accuracy is one of the most important goals of this project, and yet it is unfortunately one of the most lacking. The application produces various streams of data, of varying quality levels. The head depth and position readings are accurate to within 1-3cm, depending on conditions. As shown previously in Figure 2.12, the screen space estimation algorithm and model struggle to accurately predict the target of the operators gaze when aimed outside of a select region in the middle of the screen. While this region of the screen (Indicated in dark blue) has an error of as little as 5%, when aimed towards the edges of the monitor, the gaze tracking can be off by as much as 100%.

The measurement accuracy issues come from three primary locations within the data flow. The first is the resolution of the extracted eye images. Although a 1080p frame has over two million pixels, the region which the operators eye occupies can be contain fewer than ten thousand. Under typical conditions, the resolution of the eye window is approximately 175x75 pixels, with the center of the pupil moving only a few pixels up and down as the eye looks from the top to the bottom of

the screen. This highlights the second issue, which is the accuracy of the pupil location algorithm. Although this algorithm is fairly accurate, it has an error of plus or minus two to three pixels as lighting conditions change, making its error nearly as much as the entire vertical data range. This unfortunately poor signal to noise ratio compounds into the third section of the processing algorithm, the trained recognition model. With so much noise in both the training and testing data, this results in very poor vertical tracking accuracy, and moderately accurate horizontal tracking accuracy.

3.2.3 Portability

The current implementation of the TrackerCore was only tested in an Ubuntu 20.04 x86_64 Desktop Linux Environment with an Intel RealSense Depth Camera 435. Due to the limited amount of resources available through the duration of this project, the code was not tested on other operating systems or platforms, such as a tablet computer or Raspberry Pi. It was also not tested with other depth sensing cameras. It is not known at this time what impact this limitation had on Performance and Accuracy.

3.2.4 Racial Bias in AI and Computer Vision

Systems that rely on Artificial Intelligence and/or Computer Vision for processing images of human faces are inherently predisposed to suffer from issues relating to the race or ethnic background of their subjects. Even facial recognition systems used by large companies who make test proctoring systems have a tendency to severely under-perform when used with persons of minority origin [15]. Unfortunately due to COVID-19, the models and algorithms used by this paper could not be tested on a diverse group of subjects, and will need to be adjusted and retrained on a larger dataset.

3.3 Future Work

In order to bring this system up to the quality, accuracy, and ease of use standards that would allow it to function as a drop-in replacement for most commercial tracking systems, there is work that needs to be done. This section of the paper contains an overview of that work, and provides starting points to launch from.

Copies of all code used in this project can be found on GitHub at https://github.com/rgbd-motion-gaze-tracking/FINAL. A copy of repository as it existed at the time of submission is available from the WPI library along with copies of this document.

3.3.1 Performance

Implementing the performance monitoring system allows for much greater visibility into the various limitations of the system. As can be seen in Figure **??**, the two slowest functions are `core.trackercore.extract_eyes` and `core.trackercore.estimate_screen_pos`. The eye extraction function is slow because of its reliance on nested loops, structures which Python is notoriously slow at executing. It may be possible to rewrite this section of the application using a higher performance library, or by implementing a python extension module in C or C++. The second major source of latency is the cost of executing an iteration against the TensorFlow model. This is a more difficult area to assess potential solutions for, due to my limited knowledge of tensorflow and other neural network libraries. Using a CUDA enabled GPU or re-implementing the model using a different correlation framework may speed this up significantly.

3.3.2 Accuracy

The concerns about accuracy discussed above primarily stem from problems with camera resolution and model accuracy, neither of which can be directly addressed without replacing potentially significant components of the application. It may be worthwhile looking into other cameras in the realsense family, or other scientific grade imaging devices which might be able to capture higher resolution images of the eyes. It may also be possible to implement a more accurate pupil tracking algorithm. Be warned that the current pupil tracking algorithm was the result of months of trial and error with more sophisticated and accurate approaches such as Seonwook Park's ELG[16] and DPG[17] algorithms, of which we were unable to implement either. If you are able to get these systems working, that would dramatically increase the accuracy of this system.

3.3.3 Portability

Although the current system was only tested on Ubuntu 20.04 x86_64, it may be desirable to run this system on older/newer versions of Ubuntu, other distros, or other architectures such as a Raspberry Pi (ARM). During this project, there were issues with the binary component of Intel's `pyrealsense` library. With Ubuntu 16.04 LTS now in end-of-life status, Ubuntu 18.04 remains the only other supported LTS branch. Assuming software compatibility, porting this system to another computer should not be a monumental task, although there may be limitations with file paths that need to be addressed.

3.3.4 Racial Bias in AI and Computer Vision

It is an uncomfortable but undeniable fact that computer vision does not treat all operators equally. Due to the COVID-19 lockdown, this system could not be tested on a diverse group of people, so its effectiveness on operators with different skin tones, facial structures, eye color, etc. is unknown at this time. If permissible by the school, it may be valuable to conduct a user study testing the limitations of this system on a range of ethnic backgrounds to determine the extent of the issues.

References

[1] Tobii AB. *Tobii Pro*. URL: https://www.tobiipro.com/.

[2] Tobii AB. *Tobii*. URL: https://www.tobii.com/.

[3] Zhi Li et al. “Development of a tele-nursing mobile manipulator for remote care-giving in quarantine areas”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 3581–3586. DOI: 10.1109/ICRA.2017.7989411.

[4] B.S. Pranathi et al. “Sahayantra - A Patient Assistance Robot”. In: *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2020, pp. 1–6. DOI: 10.1109/ICCCNT49239.2020.9225444.

[5] A Kar and P Corcoran. “A Review and Analysis of Eye-Gaze Estimation Systems, Algorithms and Performance Evaluation Methods in Consumer Platforms”. In: *IEEE Access* 5 (2017), pp. 16495–16519. DOI: 10.1109/access.2017.2735633.

[6] A Duchowski. *Eye tracking methodology: Theory and Practice*. Singer, 2003.

[7] Z Zhu and Q Ji. “Robust real-time eye detection and tracking under variable lighting conditions and various face orientations”. In: *Computer Vision and Image Understanding* 98.1 (2005), pp. 124–154. DOI: 10.1016/j.cviu.2004.07.012.

[8] D Hansen and Qiang Ji. “In the Eye of the Beholder: A Survey of Models for Eyes and Gaze”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.3 (2010), pp. 478–500. DOI: 10.1109/tpami.2009.30.

[9] K Wang and Q Ji. “Real Time Eye Gaze Tracking with 3D Deformable Eye-Face Model”. In: *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), pp. 1003–1011. DOI: 10.1109/iccv.2017.114.

[10] Y Li, DS Monaghan, and NE O’Connor. “Real-Time Gaze Estimation Using a Kinect and a HD Webcam”. In: *MultiMedia Modeling* (2014), pp. 506–517. DOI: 10.1007/978-3-319-04114-8\{_}43.

[11] Kang Wang and Qiang Ji. “Real time eye gaze tracking with Kinect”. In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2016, pp. 2752–2757. DOI: 10.1109/ICPR.2016.7900052.

[12] Kenneth Alberto Funes Mora and Jean-Marc Odobez. “Geometric Generative Gaze Estimation (G3E) for Remote RGB-D Cameras”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014.

[13] Li Jianfeng and Li Shigang. “Eye-Model-Based Gaze Estimation by RGB-D Camera”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2014.

[14] M Yahya et al. “Motion capture sensing techniques used in human upper limb motion: a review”. In: *Sensor Review* 39.4 (2019), pp. 504–511. DOI: 10.1108/sr-10-2018-0270.

[15] P. Ninja. *Proctorio’s facial recognition is racist.* Mar. 2021. URL: https://proctor.ninja/proctorios-facial-recognition-is-racist.

[16] Seonwook Park et al. “Learning to Find Eye Region Landmarks for Remote Gaze Estimation in Unconstrained Settings”. In: *ACM Symposium on Eye Tracking Research and Applications (ETRA)*. ETRA ’18. Warsaw, Poland: ACM, 2018.

[17] Seonwook Park, Adrian Spurr, and Otmar Hilliges. “Deep Pictorial Gaze Estimation”. In: *European Conference on Computer Vision (ECCV)*. ECCV ’18. Munich, Germany, 2018.

Appendix A: ROS Topics Overview

In order to facilitate the integration with existing robotic systems, an implementation of the tracker core as a ROS node was created. This node implements the basic functionality of the tracker core, and exposes it through several ROS topics and basic messages. The below table provides a comprehensive overview of the topics and messages currently implemented.

Topic Name	Fields	Description

`/tracker_eyes`	`/EyePoint left`
	`float32 angle`
	`float32 x`
	`float32 y`
	`/EyePoint right`
	`float32 angle`
	`float32 x`
	`float32 y`
		Provides raw pupil and eye orientation readings for both of the operator's eyes. This information is primarily useful for debugging, or potentially passing the gaze information from a lower powered operator terminal to a system running a larger gaze tracking model, potentially a dedicated machine learning server.

`/tracker_face`	`float32 face_depth`
	`float32 face_x`
	`float32 face_y`
	`float32 face_angle`
	`float32 depth_diff`
		Provides raw face readings. The face depth is the number of millimeters away from the RealSense the operator is located. Diff reports the depth difference between the cheeks.

`/tracker_gaze`	`float32 x` `float32 y`	Estimated gaze target on screen, if this value is enabled in the TrackerCore configuration.
`/tracker_cmd`	`string command` `string value`	Change a TrackerCore configuration value. See the Control Commands section of `trackercore.py` and the command message subscriber in `tracker.py` for more information on valid commands.
`/tracker_log`	`string message`	Provides message output from the TrackerCore module. This message is primarily intended for debugging and demonstration purposes. Messages will contain ANSI format codes, which must be stripped or interpreted.

Appendix B: Example Performance Report

Below is an example report that was generated by running the `demo.py` application. Every called function which possessed the `@debug.funcperf` decorator is listed in the table below, along with the total time spent inside that function, the number of times that function was called, and the average execution time.

```
Performance Report:
|Function                                | Tot. Exec|     Calls| Avg. Exec|
|----------------------------------------|----------|----------|----------|
|core.trackercore.init                   |   1.92 s |         1|   1.92 s |
|core.trackercore.loop_once              |  13.23 s |       213|  62.14 ms|
|core.trackercore.extract_eyes           |   5.11 s |       213|  24.00 ms|
|core.trackercore.estimate_screen_pos    |   4.97 s |       213|  23.34 ms|
|core.util.log                           |  89.87 ms|         6|  14.98 ms|
|core.trackercore.geteye                 |   5.11 s |       426|  12.00 ms|
|core.trackercore.process_face           |   2.46 s |       213|  11.55 ms|
|core.trackercore.preprocess_images      | 620.43 ms|       214|   2.90 ms|
|core.trackercore.acquire_image          |  68.60 ms|       214| 320.55 us|
|core.util.scale                         |  49.22 ms|       213| 231.06 us|
|core.util.draw_poly                     |  14.67 ms|       426|  34.44 us|
|core.util.draw_line                     |   3.07 ms|       426|   7.20 us|
|core.util.crop                          |   4.95 ms|       854|   5.79 us|
|core.trackercore.debug_draw_features    | 511.41 us|       213|   2.40 us|
|core.util.sample                        |  21.76 ms|     15762|   1.38 us|
|----------------------------------------|----------|----------|----------|
```