

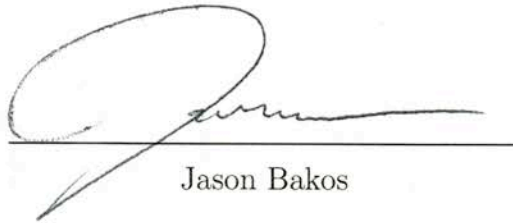
SWVL: A Custom AI-Powered Face Tracking Camera Gimbal

Alexander Anderson-McLeod, Jakub Jerzmanowski, Michael Laitarovsky, Trevor Allison, and Jagger Tanner

Submitted in Partial Fulfillment of the Requirements for
Graduation with Honors from the South Carolina Honors College

May, 2025

Approved:

A handwritten signature in black ink, appearing to read 'Jason Bakos', is written over a horizontal line.

Jason Bakos

Director of Thesis

Steve Lynn, Dean

For South Carolina Honors College

Abstract

In response to the growing demand for smarter, more responsive face tracking cameras in the post-pandemic world, our team designed SWVL, a custom AI-powered face tracking gimbal meant to address the limitations commonly encountered by the commercial models currently on the market. These commercially available gimbals come with several issues, such as frequently losing track of the person in the frame and requiring manual resets, which we sought to fix with our implementation. We designed a system with fully custom hardware and software including a 3D printed dual-axis camera gimbal driven by stepper motors, a control PCB based around an ESP32 microcontroller, and a Python-based face tracking pipeline with a custom model and MobileNetV3-Large backbone. The model communicates with an Electron UI and Python backend that work together to keep the user centered in the frame. Although it has some shortcomings, we largely achieved all our objectives and were able to produce a gimbal that is quite effective in tracking the user as they move throughout the environment. All of the code and project files for SWVL can be found on GitHub at <https://github.com/alexthecat123/SWVL>.

Contents

Description of Problem	2
Solution	3
Approach/Methodology	6
3D-Printed Gimbal Design	6
Preliminary Electrical Hardware	7
Python Backend	9
User Interface	9
Custom PCB	12
ESP32 Core	12
Stepper Motor Drivers	13
Power	13
USB Hub	14
USB to UART	14
Miscellaneous I/O	15
AI Face-Recognition Model	16
Use of Feedback	19
Potential Significance/Application	20
Appendix	21
A SWVL Command Protocol	21
B PCB Design Documents	22
C AI Model Classification Examples	24
Works Cited	28

Description of Problem

In the post-pandemic world, the daily use of webcams has increased dramatically, with applications ranging from Zoom meetings to recording online lectures. Many users would greatly benefit from a face-tracking camera that ensures they always remain in frame, without restricting the individual to a fixed position in the physical world. While AI-enabled face-tracking webcams, such as the Insta360 Link and OBSBOT Tiny, are currently available, they suffer from serious design flaws that limit their practicality in real-world applications. For instance, these cameras excel in close-range, single-subject scenarios but struggle in long-range situations or when multiple people are present in the frame. In addition, these cameras often fail to regain tracking when a subject leaves and reenters the frame, and they require a manual reset to properly begin tracking again. These cameras also have gesture recognition that can be used to adjust certain camera settings, which is highly effective but can also be inadvertently triggered by errant hand movements, leading to unexpected camera behavior. These shortcomings can be incredibly frustrating to anyone who engages in video conferencing, content creation, or any other area in which it is important to have a camera that can follow a person as they move around without requiring manual control, to the point that certain people may give up on using face-tracking cameras altogether.

Solution

This project has sought to develop an improved AI-enabled face tracking camera gimbal, called SWVL, to address the shortcomings of the cameras currently on the market.

The core of the SWVL gimbal is a 3D-printed frame that we designed to allow easy assembly of all the gimbal components, while remaining fairly compact. The frame consists of a base with a stepper motor that can rotate an upper platform 180 degrees in each direction, controlling the pan axis of the camera. This platform then has two 3D-printed risers attached to it that hold a camera sled in between them. Under the control of another smaller stepper motor mounted to one of the risers, this sled can tilt up and down 90 degrees in each direction, allowing fine control of the tilt axis of the camera.

Both stepper motors, as well as the camera itself, are connected to a custom control board based on an ESP32 microcontroller that accepts commands from the user's computer and moves the two axes of the gimbal accordingly. The control board is connected to the computer via a single cable, which carries both the gimbal commands and the camera data. Due to the power requirements of the motors, the control board must be powered externally by a separate 12V power adapter.

An Electron application runs on the host computer to allow user control over the gimbal, including USB port selection and the ability to toggle face tracking on and off. This application communicates with a Python backend that runs the face-tracking AI, developed using PyTorch. The Python program also handles sending commands to the ESP32 microcontroller to move the gimbal itself.

When designing SWVL, we put considerable thought into what types of motors to use for driving the gimbal, including brushless DC motors, stepper motors, and even brushed gear motors. Although brushed gear motors can be nice because of how straightforward they are to control and because of their holding torque, we soon eliminated them due to the difficulty in achieving fine position control with such a motor. We eliminated brushless DC motors next because, although they would provide the smoothest gimbal movement and are what are used on many commercial gimbals, they are difficult to precisely control and do not have enough holding torque to hold up our rather heavy gimbal setup. This left stepper motors, which have plenty of holding torque and fine position control, with the slight (but not incredibly noticeable) caveat of choppy position control.

Picking a camera was another difficult choice. We initially wanted to choose a 4K UHD camera for maximum picture clarity but soon realized that the processing overhead for such a camera would make it difficult to achieve reasonable tracking speeds. Furthermore, this would have required us to implement a USB 3.0 hub (discussed later), which is a rather difficult design problem that we were unsure if we could implement properly. Therefore, we decided on a [1080p Microsoft LifeCam Studio Webcam](#), which provides high-quality 1080p video, while operating on an easier to design USB 2.0 bus.

Choosing our frontend and backend software was yet another decision where we considered several options. For the backend, we considered C++ and Python due to the widespread use of both for artificial intelligence and machine learning tasks but ultimately decided on Python because of its simplified frontend integration and communication with the gimbal hardware. For the frontend, we considered several frameworks, including Electron, React, wxWidgets, and Qt. We quickly eliminated wxWidgets and Qt because of unpleasant experiences that Alex has had with them in a previous project, leaving Electron and React. We made the decision between these two using a combination of research about which one is better-documented and supported, and some feedback from friends who have used both and have a preference. Both research methods pointed to Electron as the superior option,

so that is what we chose.

The backbone of our AI is a lightweight convolutional neural network based around MobileNetV3-Large, a model that is designed for high accuracy with minimal computational overhead that is pretrained on the ImageNet dataset. This model is responsible for performing feature extraction on the input stream from the camera before passing it on to a custom head for bounding box prediction. We selected MobileNetV3-Large largely for its efficiency, considering it to be the best balance between performance and speed. Upon loading it, we prepare to use only the feature extraction layers by removing the classifier head and global pooling layers - this way, our outputs are feature maps that can be used to locate objects rather than classification scores.

Next, we pass these feature maps into a custom regression head where we perform depthwise separable convolution. We pass it through four stages where each is depthwise convolution followed by pointwise convolution, alternating between the two. Each iteration further refines the feature map to specifically finding facial features, helping the model to better understand object locations and spacial patterns. Next, the output tensor is flattened, resulting in a vector of size 188,160 that we then pass through two fully connected layers. In the first, the vector is compressed into a latent space of size 4,096 to reduce dimensionality while retaining feature information. We then apply both Dropout and LeakyReLU to aid in regularization and gradient flow before passing it into the next fully connected layer. This layer outputs bounding box information in a grid format, encoding information about the center coordinates, width, height, and confidence score for each of B predicted boxes per cell in an SxS grid.

Approach/Methodology

Given that it is difficult to write gimbal software without hardware to test it on, the first component of the SWVL system that we implemented was the hardware itself.

3D-Printed Gimbal Design

We designed the frame of the gimbal in [Onshape](#), an online cloud-based CAD platform that is similar to Autodesk Fusion 360, but free to use. The design consists of a circular base unit, a circular pedestal that sits on top of the base, two vertical risers that attach to the pedestal, and a camera sled that mounts between the risers. A large NEMA-17 stepper motor mounts into the cutout in the base to rotate the pedestal, which forms the pan axis. After the two risers and sled are installed into the pedestal, a smaller NEMA-8 stepper motor is bolted to the side of one of the risers to drive the sled, forming the tilt axis. The shafts of the motors press-fit into holes in the bottom of the pedestal and the side of the sled, eliminating the need for more complex mounting mechanisms. The distance between the risers and the depth of the sled are both designed to be significantly larger than necessary for the Microsoft webcam to allow the potential use of larger cameras in the future.

These components were then exported as STL files and 3D-printed on a Bambu Lab X1 Carbon 3D printer using PLA filament. This has proven to be suboptimal due to heating from the motors deforming the motor shaft holes, so future revisions of the design should be printed in higher-temperature filaments such as PETG or ABS.

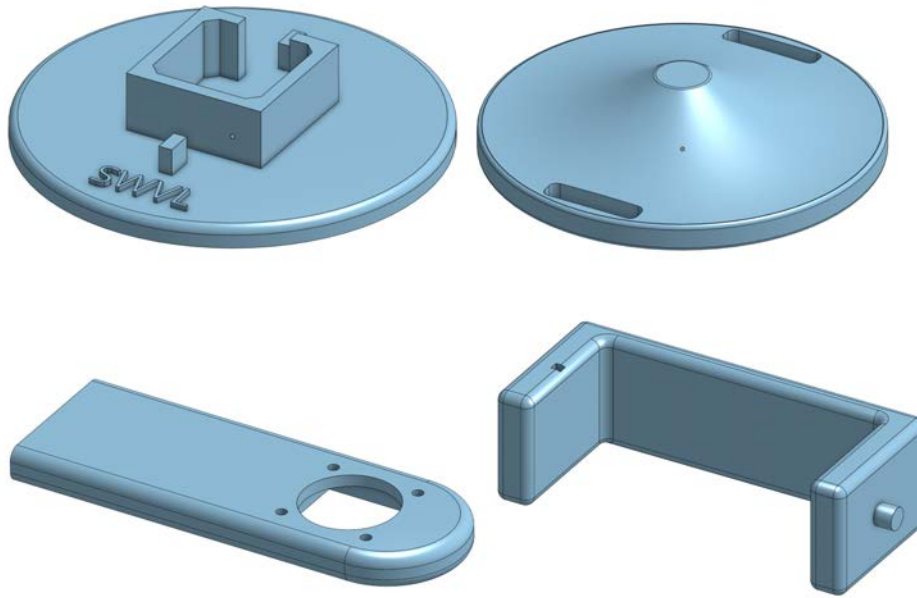


Figure 1: Starting at the top-left and going clockwise, the SWVL base, pedestal, sled, and one of the risers.

Preliminary Electrical Hardware

Once the frame was complete, it was time to begin work on the electronics. First, the [NEMA-17](#) and [NEMA-8](#) motors were installed in the frame, and then a test circuit was constructed using an Arduino Uno microcontroller. The ultimate plan was to design a fully-integrated custom PCB based around an ESP32 microcontroller, but it was important to validate the design with less expensive hardware that we already had on-hand before putting in the time, effort, and expense to design and fabricate a board.

This test setup consisted of the Arduino Uno with a stepper motor driver shield stacked on top, which was populated with two A4988 stepper motor driver modules, one for each motor. Each of these accepts a direction and step signal from the Arduino, among other signals that are less important here, which we can use in combination to control the motor's direction, speed, and fine position. The motors require 12V, while the Arduino runs on 5V, so a 12V DC power adapter was connected to the stepper driver shield as well. Finally, each motor was plugged into the connector for its corresponding stepper driver, completing the

Arduino setup.

Next, we needed to write code to run on the Arduino that would accept commands from the host computer over USB and move the motors according to whatever commands it received. This program was written in C++ using the Arduino IDE, and the command set used to communicate with the gimbal is documented in the Appendix.

At this point, it was almost time to test the motors. The final thing that we needed to do was to attach the camera to the gimbal, to ensure that the motors would be able to carry its weight. Initially, we simply hot-glued it to the center of the sled. We then sent the gimbal some commands over serial, and while the commands executed properly, it was clear that the tilt axis did not have enough torque to carry the weight of the camera. We decided to remove the plastic enclosure from the camera to save weight, before gluing it back onto the sled again, and this was enough of a weight savings to allow the gimbal to move smoothly along both axes.

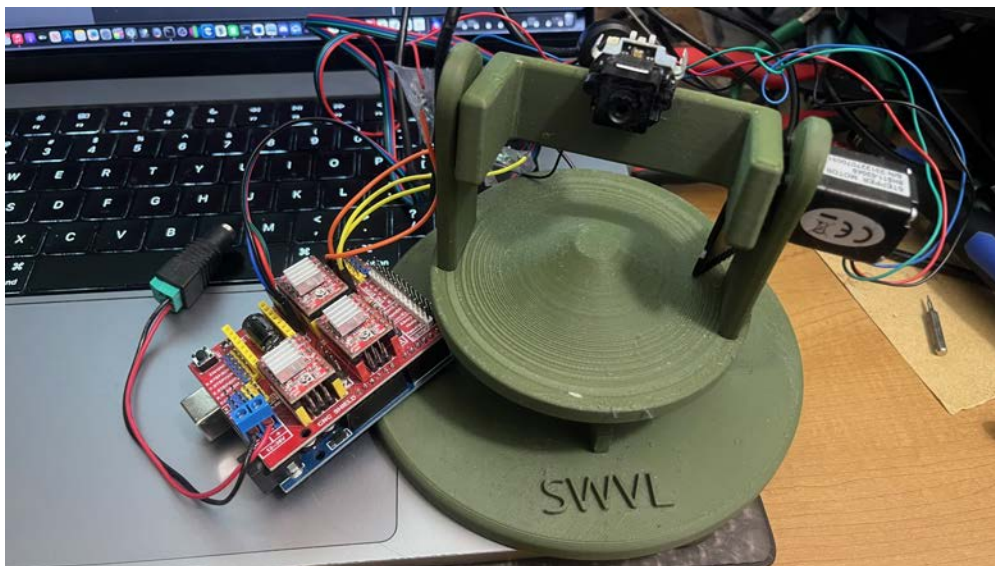


Figure 2: The prototype SWVL gimbal design.

Python Backend

Next, it was time to implement a proof-of-concept face-tracking demo using an off-the-shelf algorithm. We used Python's OpenCV package to read data from the camera, and used the OpenCV Haar Cascade facial classifier for off-the-shelf facial recognition. We wrote code to get the location of any faces in the frame from the classifier, isolate the largest one, place a bounding box around it for the user's viewing pleasure, and compute the X and Y distances between the center of the frame and the location of the face. If the face deviated from the center of the frame by more than a certain amount, then the program would send pan and tilt commands to the gimbal's Arduino to attempt to center the face in the frame. This demo was actually quite effective, and gave us the encouragement we needed to pursue the project further.

User Interface

The next step was to create a user interface for controlling the gimbal instead of relying solely on a Python program that was launched from the command line. The Python program would still form the facial recognition backend, but it would need to communicate with a user-friendly frontend of some kind, which we were planning on designing in Electron. Fortunately for us, this frontend could be quite simple, given that all it needed to do was provide a preview of the video from the gimbal, provide a means by which the user could select which USB port the gimbal is connected to, and include an "Enable Face Tracking" toggle switch.

Despite appearing simple on the surface, this actually proved to be quite difficult to implement. The primary challenge was handling all the potential errors that could occur given that we are using actual hardware that could fail, become disconnected in the middle of use, and so on. Additionally, there were challenges such as only showing the user USB connection options for devices that might actually be gimbals, refusing to connect to any

device that does not end up being a gimbal, handling errors on the Python backend, and more. These sorts of error handling routines ended up forming the vast majority of the Electron JavaScript code, with comparatively small amounts being dedicated to the user interface itself.

The communications between the Electron app and Python were handled using a Python package called Flask, a web framework that allows the two applications to exchange information using HTTP requests. An example of this communication channel in use can be seen during the USB connection process, in which the Electron app passes the name of the USB port that the user wants to connect to down to the Python program, and then the Python program returns an appropriate response depending on whether the gimbal connection succeeded or failed.

Other than the addition of Flask for communications and multithreading to run Flask in one thread and the tracking code in another, the Python code remained relatively unchanged from the previous demo version, with the only major modifications being a check of the gimbal's identity byte before connecting to ensure that it is actually a gimbal and a flag that can be used to enable and disable tracking instead of having it enabled all the time.

Given that the Electron app needs to be able to launch the Python program when it starts, we used a program called Pyinstaller to convert our Python program into an executable binary that the Electron app can launch through the operating system when it starts up. Pyinstaller makes sure to include all of the Python program's dependencies, such as Flask and OpenCV, in the executable, allowing for an easy cross-platform solution that does not require manual installation of packages and does not even require the end user to have Python installed on their system.

After some testing and debugging, we found that our new frontend worked very well, with one caveat that we have still been unable to overcome. A camera device can only be opened by a single process at a time, preventing Python from opening the camera for face tracking and Electron opening it simultaneously for displaying a preview to the user. This also means

that the user would be unable to use the camera in any other applications, which is a major shortcoming that renders SWVL impractical as anything other than a proof of concept. We have investigated several methods to split the camera feed into multiple virtual devices that can then each be opened by separate applications, including V4L2 on Linux, but we have been unable to get these methods to work properly. Other devices are able to perform such a task, so clearly it is possible, but we have been unable to achieve it. Of course, using OBS to split the feed is always an option, but we want a command line-based solution that can be directly configured by our software and that is much more compact than something like OBS, which is designed for recording and streaming in addition to creating virtual cameras.

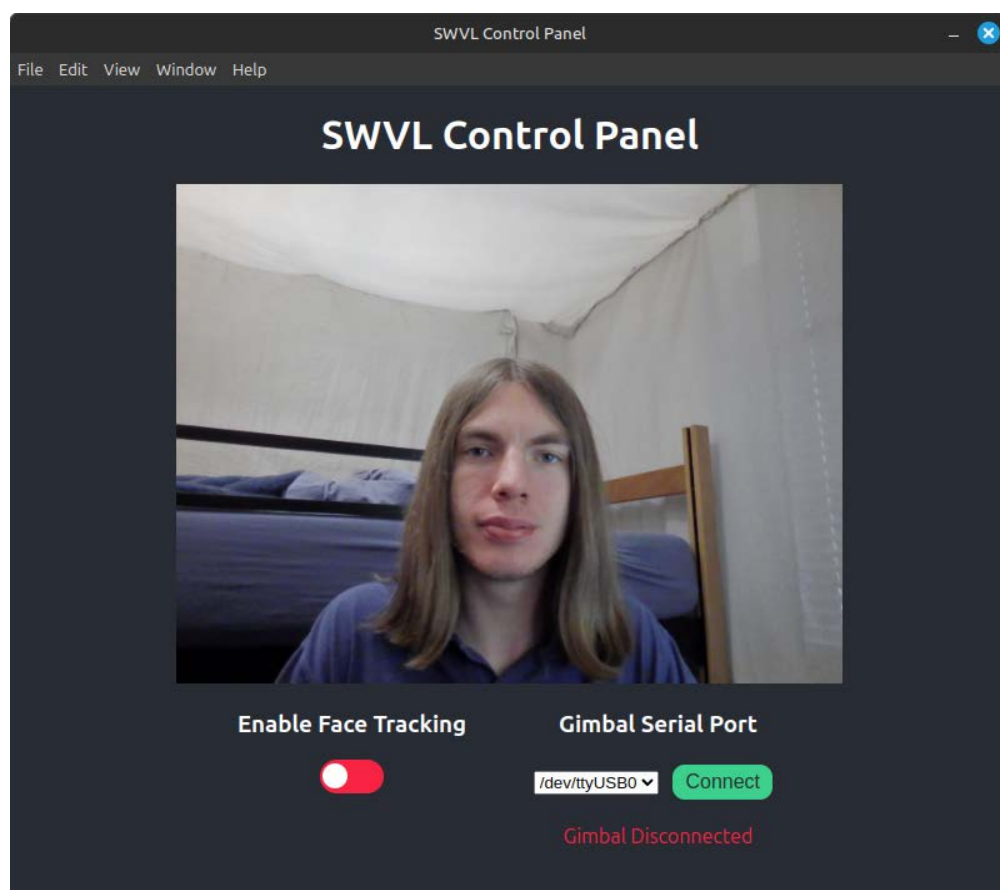


Figure 3: The Electron-based SWVL user interface.

Custom PCB

After reaching this point, there were two major aspects of the project left to complete: a custom PCB to hold all of the electronics and the custom AI face-tracking algorithm.

The custom PCB was designed in [EasyEDA](#) and fabricated using [JLCPCB](#). The board design can be split into six main logical components:

- ESP32 Core
- Stepper Motor Drivers
- Power
- USB Hub
- USB to UART
- Miscellaneous I/O

ESP32 Core

The board design is centered around an ESP32 microcontroller, which replaced the Arduino that was used in the prototype design. This change was made because the ESP32 is the same price as the Arduino, while being significantly faster (240MHz and dual-core versus 16MHz and single-core), including built-in WiFi and Bluetooth, containing significantly more flash memory and RAM for program use, and having numerous hardware peripherals that offload certain tasks from the CPU itself, while being compatible with the original Arduino code with very few modifications. Given that the original code ran on an Arduino without issue, we are not utilizing any of these benefits of the ESP32 here, but they are nice to have for potential future changes to the software. The ESP32 (the large module in the center on the right side of the board in Figure 4) is also surrounded by several support components, such as bypass capacitors, transistors that put it in bootstrapping mode based on certain serial commands, and so on.

Stepper Motor Drivers

The custom PCB uses the same A4988 stepper drivers as the prototype board, except this time they are integrated directly onto the board instead of being plug-in modules. Each of the two A4988s is surrounded by about a dozen support components, all capacitors and resistors, to configure the chips to our needs. See the A4988 datasheet in the References section for more information on each component and what its purpose is.

Each motor plugs into its corresponding header next to its A4988, and the large blue potentiometer between the A4988 and each header serves as a current adjustment for each motor. This potentiometer gets set to provide the motor with the lowest amount of current that allows it to have enough torque to move and hold the gimbal. Setting it too high can cause the A4988s to overheat. To minimize heating, each A4988 also has a small heat sink installed on top of it.

Power

The ESP32 is a 3.3V microcontroller, while the USB bus runs on 5V. Furthermore, the stepper motors run on 12V, so there are 3 voltages that our board needs to function.

The 12V for the motors is supplied by an external DC power adapter that plugs into a barrel jack on the board; we considered generating this voltage from the 5V USB voltage using a boost converter, but decided against it after realizing that this would draw more current than the USB specification allows.

The 3.3V bus for the ESP32 is generated using an AMS1117-3.3V linear voltage regulator that takes the 5V USB voltage from the host computer and regulates it down to 3.3V.

There are several filter capacitors to ground on the output of the regulator, as well as all around the board near the power inputs of each chip, to suppress voltage spikes on all the power rails. In addition to these filter capacitors, each rail has one or more large storage capacitors, most notably the large capacitors on the 5V and 12V rails next to the camera's USB port.

USB Hub

In order to allow both the ESP32 and the camera to communicate with the host computer over a single USB cable, it was necessary to implement a USB hub on the SWVL control PCB. Due to the immense signal integrity challenges that arise when implementing a USB 3.0 or greater hub, we chose to implement a USB 2.0 high-speed (480Mbps) interface, which is fast enough for a high-bitrate 1080p camera, while still being manageable to design.

We settled on a CH334 USB hub chip due to its low cost, thorough datasheet, and availability in JLCPCB's PCB assembly parts library. Due to the high speed of the USB bus, signal integrity was crucial when designing the USB circuitry, meaning that we had to carefully plan the placement and trace routing for each component around the USB hub. After connecting everything as dictated by the datasheet and our particular use case, we had to check it against the USB specification to ensure that everything was within spec. After determining that everything seemed to match, we had to perform the final step of routing the USB traces themselves from the USB ports and ESP32 to the hub chip itself. These traces are very sensitive, and must be routed side by side as a differential pair with a very specific differential impedance. Furthermore, it is highly discouraged to place vias anywhere along these traces, so the requirement to route them entirely on the top side of the board was an additional challenge. Eventually, we were able to route the traces to all three USB devices (the host computer, the camera port on the top edge of the board, and the ESP32) properly.

USB to UART

The ESP32 receives programs and communicates with the host computer via a UART serial interface, but it is necessary to convert this to USB in order to pass it through the hub to the host computer. This was accomplished through a CP2102 USB to UART chip, which we chose because of its reputable manufacturer (Silicon Labs), low price, thorough datasheet, and ease of design. The chip requires very few support components, so after adding six

capacitors and resistors, routing a differential pair from the chip to the USB hub, and routing RX and TX traces to the ESP32's UART pins, this portion of the design was done.

Miscellaneous I/O

Given that the cost of common components, particularly SMD components, is minimal, we decided to add a few additional I/O devices to the board.

First, we added an EN (reset) and a BOOT button to the board, which allow the resetting of the ESP32 and the ability to put it into bootloader mode for the sake of debugging.

We also added a speaker so that the user can receive auditory feedback from the gimbal in the form of beeps when it connects to the host and when tracking is enabled or disabled. The speaker is driven by a 2N2222 transistor connected to one of the ESP32's GPIO pins.

Finally, we added a WS2812 addressable RGB LED for visual feedback to the user. Currently, the LED is red when the gimbal is idle and turns green when tracking is enabled, but more complex behavior could easily be enabled given that this is an RGB LED capable of producing 16 million colors. The LED's data line is connected directly to one of the ESP32's GPIO pins and the LED is powered by 5V.



Figure 4: 3D rendering of the SWVL PCB with all components populated.

AI Face-Recognition Model

Of our options for the design of the AI component, we first ruled out multi-layer perceptron (MLP), as the model needed to be feature invariant. While MLPs are effective for the purposes of classification, they are suboptimal for image detection or understanding spatial structure.

In our first design, we applied convolutional layers followed by global average pooling. This approach computed the mean value across each channel to output a feature vector. However, while global average pooling is well-suited for classification purposes, it is not effective for object detection purposes; it compresses the feature map to a single value, which loses spatial information that is required for purposes like bounding box object detection. To solve this, we removed the global pooling and tried flattening the feature maps into a one-dimensional vector, which allowed us to retain the spatial information needed and produced a notable improvement in bounding box prediction. However, it was very slow; convolution is very computationally expensive and performing it on a large scale had a clear impact on performance. To fix this, we implemented depthwise separable convolution, a technique that uses two convolution operations to significantly reduce the number of total computations. The first operation is depthwise convolution, where we split the channels into groups and then perform convolution on all channels in each group in parallel. The second is pointwise convolution, in which we perform convolution across all channels at once.

Our next adjustment was an improvement to our loss function. Originally, it produced a singular vector of length 4 that included parameters for the predicted width, height, x, and y values for a bounding box. However, this forced the model to make a prediction for every grid cell, even if there was no face present. We accounted for this by adding a fifth parameter, a confidence score, which is a measure of how confident the model is that there is a face present in the cell. This allows us to improve training accuracy by ignoring boxes with low confidence scores.

We originally tried to train our own custom backbone from scratch but would not have

been able to get the results we wanted in time. We trained it for a week on 2 NVIDIA GeForce RTX 3090s and believe with more time we could have gotten competitive results, but due to the time constraints we decided to instead use a pretrained backbone and settled on MobileNetV3-Large, which gave us satisfactory results in terms of accuracy and efficiency.

We needed to find a dataset that was sufficiently diverse, as well as containing bounding box data rather than just labels for image classification tasks. Some of the qualities we were looking for were significant variations in lighting, gender, age, race, size in frame, and angle. After spending a while looking through potential datasets, we settled on the Face-Detection Dataset uploaded by Fares Elmenshawii on [kaggle.com](https://www.kaggle.com). This dataset is of sufficient size, consisting of 16.7K images originally scraped from Google Open Images. It contains bounding box data for the faces in each image, as well as having substantial variation in all the categories we were interested in.

In addition to the steps mentioned above, training the model took several iterations before we were able to meet our requirements. We initially had issues where the bounding box would always be placed at the center of the frame because this always minimized the loss function, and after fixing this we had other problems where faces could only be detected when very close to the camera and when pointed directly at it. We were able to solve both problems to a certain extent, although the model still struggles to recognize faces at extreme angles or extreme distances. Furthermore, Alex tested the model on images of his cat, Little Puss, and discovered that it classified his cat's face as a human face, so changes needed to be made to remove this behavior as well. See the Appendix for samples of faces being recognized and failing to be recognized by the model.

After we completed all of this work, we were able to test the final gimbal design, and found that it was quite effective. Although it could not detect faces at extreme angles or distances, the gimbal was very effective at following any face that it was capable of detecting, and was still capable of detecting faces under the vast majority of conditions.

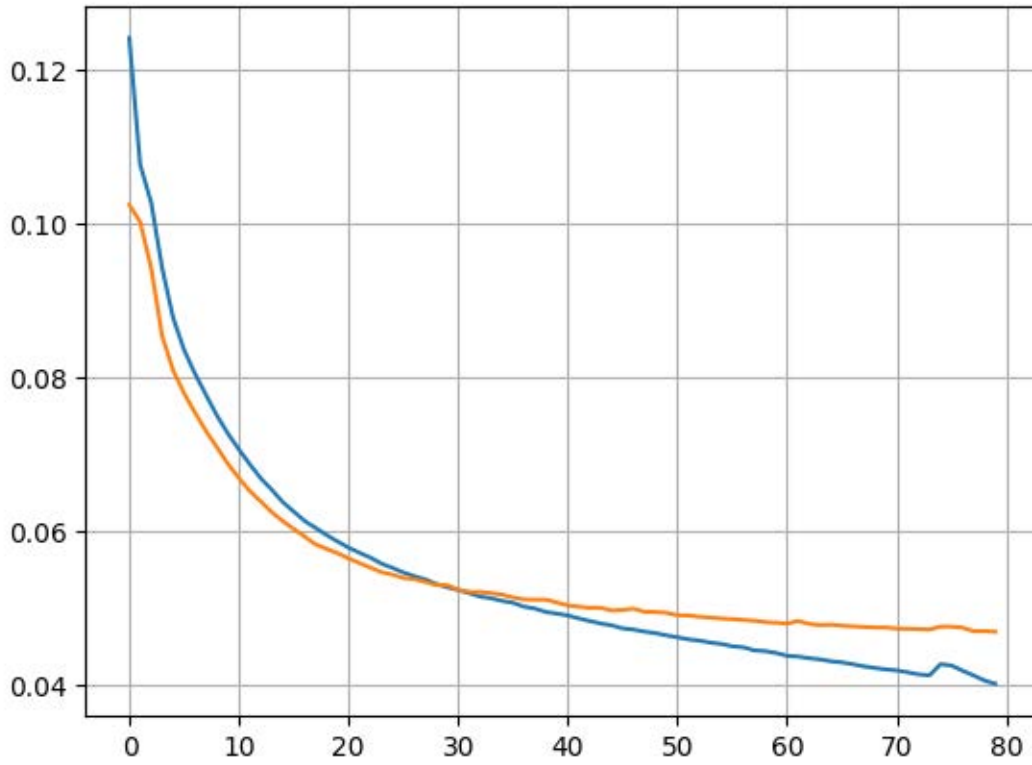


Figure 5: Loss curve of the model. Blue: Training Loss, Orange: Validation Loss.

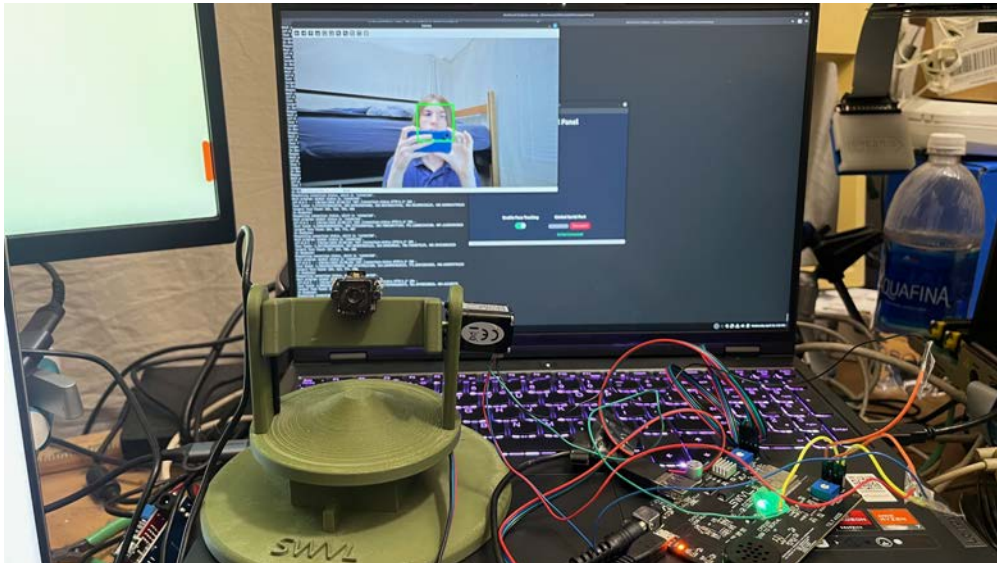


Figure 6: The final SWVL gimbal hardware and software tracking Alex's face. Notice that the bounding box is green, indicating that the gimbal has successfully moved to center his face within the frame.

Use of Feedback

We split up the workload for the SWVL project, with Alex working on the hardware, the low-level software, and the user interface, and everybody else working on the AI aspect. We would regularly meet or exchange videos to share our progress with each other, and we would provide feedback on whatever our other team members had produced. For instance, the AI team would send Alex their latest models, and he would run them through a test script that he wrote to determine what their strengths and weaknesses were. Alex would provide appropriate feedback, and they would send him an improved model in a few days that would attempt to address the feedback from the previous one.

Potential Significance/Application

Our model can accurately and efficiently detect and track a moving face in a live video feed. Although it may only match their performance in its current state, with further training and fine tuning, we believe it has the potential to outperform the options currently available on the market, while still coming in at a lower price. After this tuning, our solution would not only make AI face tracking more effective, but also more accessible to a larger group of people. Furthermore, our entire design is open source, a key benefit over other current offerings that allows others to take our work and improve upon it.

There are two primary changes that would be important to make in a future iteration of the design. The first would be to more finely tune the facial-detection model to better detect faces from extreme angles and distances. This is something that other gimbals struggle with as well, and if we were able to improve this capability, it would make our solution superior to those other offerings. The second would be to miniaturize the hardware, allowing the device to come in at a size and weight that is more comparable to the other offerings on the market. This would probably require switching from stepper motors to brushless motors and performing custom manufacturing for some of the small parts, so it may not be practical for a small hobbyist project like this unless we were putting it into commercial production.

Appendix

A SWVL Command Protocol

The SWVL gimbal is controlled via a simple command protocol sent over the serial connection. Table 1 provides a complete list of these commands and their functions.

Table 1: Table of gimbal commands.

Command String	Purpose
ID	Identify. Returns an “identity byte” back to the host computer. Used to determine whether or not the connected device is actually a gimbal.
DS	Disable Steppers. Disables the stepper motors (sets holding torque to zero) to prevent motor overheating and excessive power draw. Executed by the host program when tracking is disabled.
PR <steps>	Pan Relative. Pans the gimbal by <steps> stepper motor steps, where <steps> is a 16-bit signed integer. Each step is 1.8 degrees, and sign represents direction.
TR <steps>	Tilt Relative. Tilts the gimbal by <steps> stepper motor steps, where <steps> is a 16-bit signed integer. Each step is 1.8 degrees, and sign represents direction.
GP	Get Pan Angle. Returns the current pan angle as a 16-bit signed integer.
GT	Get Tilt Angle. Returns the current tilt angle as a 16-bit signed integer.

B PCB Design Documents

The following figures show the schematic diagram and the PCB layout for the SWVL control board.

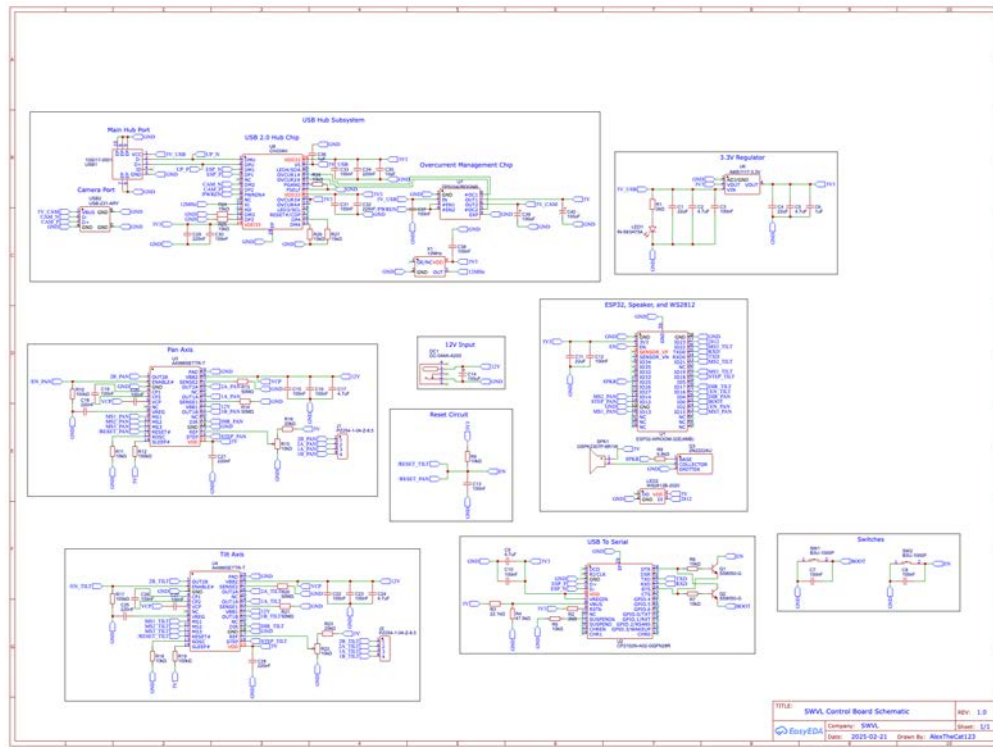


Figure 7: Schematic of the SWVL control PCB.

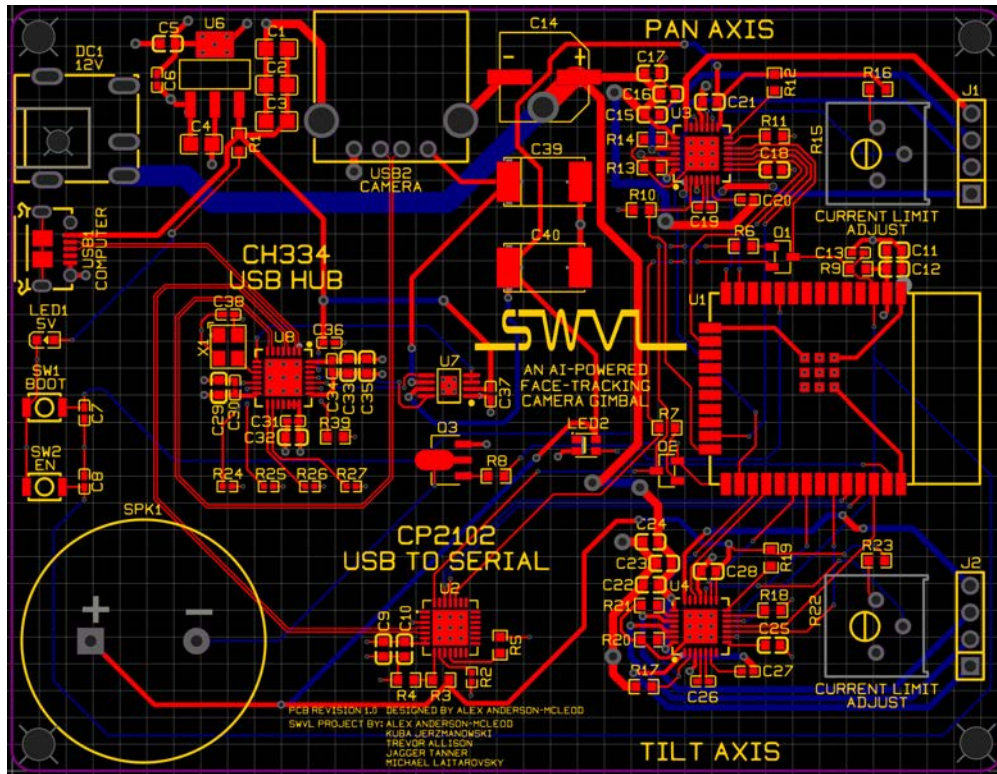


Figure 8: Layout of the control PCB. Note the differential pairs for the USB ports, and other component placement and trace routing decisions for the sake of signal integrity in sensitive areas.

C AI Model Classification Examples

The following figures demonstrate both the capabilities and limitations of the SWVL facial-recognition model under various conditions.



Figure 9: The model successfully recognizes Alex's face, even when his hair is blowing in front of it, illustrating its ability to successfully find faces that are partially-obstructed.

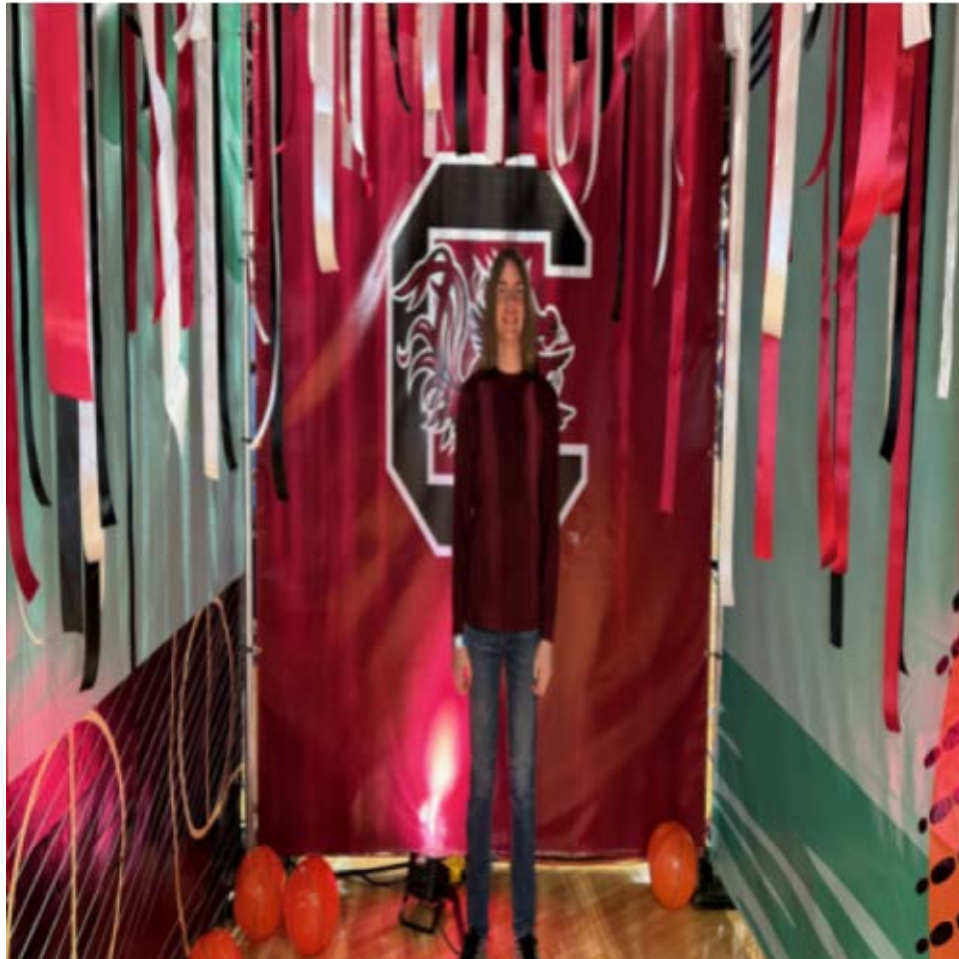


Figure 10: The model fails to find Alex's face when at a distance of approximately 15 feet, which is one of its shortcomings. Fortunately, most people are not going to be using this type of gimbal at such a distance, so this may not be a significant issue.



Figure 11: Despite the rather significant (roughly 45°) angle of his head, the model still successfully recognizes Alex's face. Although this may seem like adequate performance, we were hoping to be able to detect faces that were angled as far as 90° away from the camera, which we are currently not able to accomplish.



Figure 12: Alex discovered that the model would often detect the face of his cat, Little Puss, as a human face. However, as seen in these two images, we were able to fix this, in part by increasing the size/resolution of the image tensor that was fed to the model (notice the lower resolution of the left-hand image compared to the right-hand image).

Works Cited

- [1] <https://www.insta360.com/product/insta360-link>
- [2] <https://www.obsbot.com/obsbot-tiny-4k-webcam>
- [3] https://www.pololu.com/file/0j450/a4988_dmos_microstepping_driver_with_translator.pdf
- [4] <https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf>
- [5] <https://cdn-learn.adafruit.com/assets/assets/000/131/435/original/CH334DS1.PDF?1721660148>
- [6] <https://www.kaggle.com/datasets/fareselmenshawii/face-detection-dataset/data>
- [7] Howard, A., Sandler, M., Chu, G., Chen, L. C., Chen, B., Tan, M., ... & Adam, H. (2019). Searching for MobileNetV3. arXiv preprint arXiv:1905.02244.
- [8] Chollet, F. (2017). Xception: Deep Learning with Depthwise Separable Convolutions. arXiv preprint arXiv:1610.02357.