Gesture-Controlled AI Assistant for Intraoperative Imaging (GCAI)

Edwin Tembo Anjay Sakuru Jayden Chen

Abstract

We present the Gesture-Controlled AI Assistant for Intraoperative Imaging (GCAI). This tool harnesses gesture recognition for contactless navigation of medical images during surgical operations. Traditionally, the sterile environment of operating rooms has posed unique challenges to surgeons' ability to interact with, navigate, and manipulate medical images during intraoperative imaging. Surgical staff have often depended on communication with imaging operators in an external control room for image navigation. To improve usability and preserve operative flow, we propose GCAI that uses gesture recognition to zoom and pan both static medical images and dynamic medical imaging feeds. This enables contactless image navigation directly from the operating room for intraoperative imaging applications.

1. Introduction

The global intraoperative imaging market has been experiencing significant growth, driven by the increasing adoption of minimally invasive surgeries and technological advancements in imaging modalities. As of 2023, the market was estimated at USD 3.44 billion, with projections indicating a compound annual growth rate (CAGR) of 6.70% from 2024 to 2030. This growth is fueled by the rising prevalence of chronic disorders, particularly among the geriatric population, which is expected to reach one in six people globally by 2030, according to the World Health Organization (WHO). Intraoperative imaging technologies, such as intraoperative ultrasound (iUltrasound), intraoperative magnetic resonance imaging (iMRI), and C-arm systems, have become crucial in enhancing surgical precision and improving patient outcomes.

In this context, GCAI was developed as an AI-enabled intraoperative imaging tool designed to recognize hand gestures for navigating and manipulating medical images in real time. This project contributes to the advancements in intraoperative imaging by providing a more intuitive interface for surgeons to navigate medical images that does not compromise sterility. Ultimately, this enhances surgeons' focus and ability to perform complex procedures with greater accuracy and efficiency. The integration of artificial intelligence and gesture recognition technologies in surgical settings aligns with the broader trends of technological innovation within the intraoperative imaging market. This continues to evolve with the introduction of portable devices, augmented reality, and multimodal imaging approaches.

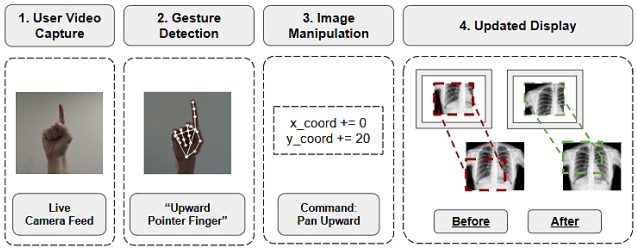


Figure 1.1: GCAI Image Manipulation System. A live video feed of the user is streamed **(1)**. If a hand is detected, GCAI locates various landmarks to determine which gesture is shown **(2)**. Based on the gesture, a corresponding image manipulation is performed **(3)**, updating the image displayed to the user **(4)**.

Despite the promising market trajectory, challenges such as the high cost of advanced imaging systems and the increasing demand for refurbished equipment may impact market expansion. Regulatory policies also play a crucial role in ensuring the safety and efficacy of intraoperative imaging devices, influencing both accessibility and innovation in this vital segment of the healthcare industry. For this, GCAI was designed prioritizing reliability, patient privacy, and versatility to enable easy integration with existing medical environments.

2. Background

Several key considerations were made in the initial design of the GCAI system. To prioritize adaptability and integration across different systems and devices, a web-based interface was chosen. Additionally, to prioritize patient privacy, portability, and to avoid network latency, on-device deployment of the gesture recognition software was chosen.

Development began with architecting the core functionalities of the GCAI system. To improve consistent detection for diverse users, GCAI's gesture detection functionality was built to leverage the MediaPipe Hands landmark detection model that accurately identifies the key points on the user's hand. This design helped ensure that GCAI would be functional in a wide variety of settings and with a diverse array of users with different characteristics such as handshapes and skin tones.

Intuitive hand gestures were chosen to improve the user experience and reduce the learning curve. Thus, gestures were created to replicate traditional gestures a user would use when interacting with a touch screen display to zoom and pan an image. Gestures were also selected so that they could be performed efficiently and in a limited space given the complex landscape of operating rooms.

To demonstrate the system's versatility, the Raspberry Pi 5 Model B (8GB) was chosen to be paired with a compatible USB camera module to test the GCAI system. Overall, this hardware configuration confirmed GCAI's potential for on-device deployment for reducing security concerns and improving adaptability and portability in a wide variety of surgical contexts (medical centers, field hospitals, etc.). The modular design also allows for future hardware updates, accommodating advances in AI-driven imaging.

Finally, an interactive front-end using HTML and CSS was designed for easy integration across different devices and systems. This design prioritized intuitive user experience and functionality.

Overall, these hardware and software considerations proved a crucial step towards making GCAI a robust and responsive tool to enhance intraoperative imaging while maintaining compactness, sterility, and ease of use.

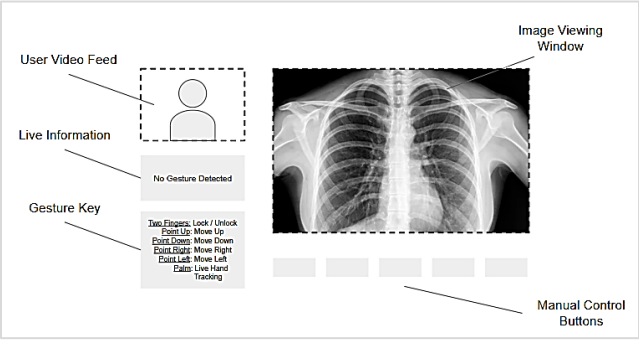


Figure 1.2: Sample User Interface Design for GCAI. Key components include a user video feed to display the captured video of the user and an image viewing window for the medical image. The displayed medical image can be either a static image or a live feed from an intraoperative imaging device. Manual control buttons are present as a failsafe.

3. Literature Review:

Sterile Processing and the Operating Room: Why Patient Safety Can't Be Rushed [25]: The article highlights the critical partnership between the Operating Room (OR) and the Sterile Processing Department (SPD) in ensuring patient safety during surgeries. It underscores the importance of empathy and teamwork, emphasizing that SPD's role extends far beyond "cleaning." Proper sterile processing, a meticulous 4.5-hour cycle involving decontamination, inspection, sterilization, quality control, and storage, prevents surgical site infections (SSIs), which account for 20% of healthcare-associated infections. Respecting this process as an investment in patient safety enhances surgical outcomes and reinforces collaboration between the OR and SPD. Future discussions will explore related topics like High-Level Disinfection.

Intraoperative Imaging Market Size & Trends [26]: The intraoperative imaging market, valued at $3.44 billion in 2023, is projected to grow at a 6.7% CAGR through 2030. Growth is driven by advancements in technology, increased adoption of minimally invasive surgeries, and the rising geriatric population. Key technologies include iUltrasound, C-arm, and iMRI, enabling precision in neurosurgery, orthopedics, and other specialties. Innovations like real-time imaging, 3D technology, and portable devices are fueling adoption. North America dominates the market, while Asia Pacific shows the fastest growth. Challenges include high equipment costs and regulatory complexities. Major players, including GE Healthcare and Siemens, focus on R&D, acquisitions, and expanding product lines.

Voice or Gesture in the Operating Room [29]: The study, *Voice or Gesture in the Operating Room*, explores the use of voice and gesture-based controls for hands-free interaction with imaging systems in sterile environments, specifically during surgery. This research underscores key considerations for our AI-enabled intraoperative imaging tool that uses hand gestures for navigating and manipulating medical images.

The findings indicate that each modality—gesture and voice—offers distinct advantages depending on the task and context. Gesture-based control allows for continuous interaction, such as zooming or rotating images, but can be challenging when a surgeon's hands are occupied or when specific gestures interfere with surgical movements. In contrast, voice commands, which are generally more accurate and less physically constrained, work well for discrete actions, such as changing modes or initiating certain image manipulations. However, voice control may be limited in performing fine, continuous adjustments.

Microsoft Research: Touchless Interaction in Medical Imaging [27]: The article discusses Microsoft Research Cambridge's innovative use of Kinect for Windows in surgery, enabling touchless control of medical imaging via hand gestures. Led by Helena Mentis, the project aims to improve precision and efficiency in procedures like stent placement and neurosurgery by allowing surgeons to manipulate CT and MRI images without breaking sterility. Tested in hospitals like St. Thomas' NHS Trust, the system enhances surgeons' control while reducing operative time and infection risk. We seek to build on this work by using more compact, intuitive gestures rather than the larger arm movements employed by this existing system. Additionally, our work reduces dependency on external devices like the Microsoft Kinect.

GestSure [28]: This system enables surgeons to navigate MRI and CT scans using sterile hand gestures in 3D space, bypassing the limitations of non-sterile keyboards and mice. This eliminates delays caused by scrubbing out or relying on nurses, saving valuable time and costs—up to $620 per scrub-out. The system utilizes the Microsoft Kinect and can be integrated into existing operating room setups. By providing quick, sterile access to imagery, GestSure reduces the risk of surgical site infections, improves efficiency, and enhances patient outcomes. We seek to build on this work by reducing the complexity of the gestures employed and eliminating the need for external devices like the Microsoft Kinect, thus making gesture control more intuitive for surgeons.

4. Methodology:

4.1. System Design:

Figure 4.1 provides an overview of the GCAI gesture to image manipulation pipeline. When the application is activated, the video feed from the webcam is captured using the OpenCV library. Each frame is processed using the MediaPipe Hands landmark detection model which outputs the coordinates of key landmarks on the user's hand. These coordinates are classified using a Feed Forward Neural Network deployed with the TensorFlow Lite (TFLite) framework. This neural network determines which gesture the landmarks represent. Based on the detected gesture, the corresponding image manipulation is executed by adjusting the CSS coordinates in the application's front end (see **Table 4.1:** Gesture Key; and **Appendix A:** Example Gestures).

If the detected gesture is "Open Palm," the "Free-Range Panning" mode is enabled, requiring an additional step of processing. In this mode, changes in landmark coordinates between subsequent frames are converted to changes in the CSS coordinates of the image to perform real-time hand tracking to pan the image. Given the use of the TFLite framework, the GCAI system can be easily deployed locally on a device, such as the Raspberry Pi 5 8 GB RAM with a compatible camera.

4.2. Model Training:

4.2.1 Data collection:

After searching the internet for appropriate images to train the model that classifies gestures, it became apparent that collecting our own data may be more beneficial and without licensing restrictions.

Eleven gestures were trained by running an inference loop continuously for at least five minutes, one gesture at a time. During this period, the subject moved farther and closer to the camera randomly while slightly changing the gesture's orientation towards the camera. Lighting conditions were also changed to try and replicate conditions from darker to brighter light intensities. This resulted in a csv file with 158,910 data points, each with 63 covariates and 1 label corresponding to the hand gesture.

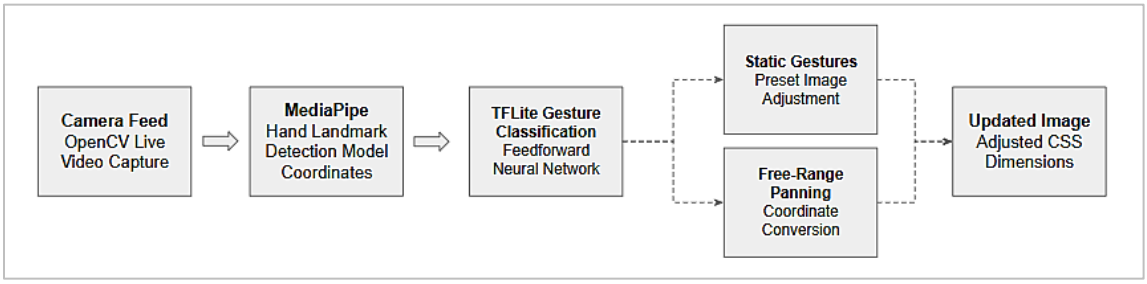


Figure 4.1: Gesture to Image Manipulation Pipeline. This diagram shows each step from video capture to the image manipulation shown on the display. Important steps involve landmark detection and gesture classification.

The data was collected over multiple collection periods, with each taking approximately 3-4 hours.

Table 4.1: Gesture Key

Gesture	Image Manipulation	Gesture Type
Two Fingers	Lock / Unlock Image	Static
Palm	Free-Range Panning	Dynamic
Upward Pointer Finger	Directional Panning - Up	Static
Downward Pointer Finger	Directional Panning - Down	Static
Leftward Pointer Finger	Directional Panning - Left	Static
Rightward Pointer Finger	Directional Panning - Right	Static
V-Shaped	Zoom In	Static
Pinched	Zoom Out	Static

See Appendix A for Corresponding Example Images

The collection process involved streaming the user's webcam images through an inference loop with the MediaPipe Hands landmark detection model [19]. The output of each loop was composed of three arrays, each corresponding to the MediaPipe model's 21 hand landmarks in 3-D space, x-, y- and z-coordinates. The x-dimension and y-dimension represent the plane perpendicular to the camera sensor and consequently the screen display. The z-coordinates represent approximate distance from the camera sensor and thus the display in our implementation. The three coordinate arrays (x, y, and z) were concatenated into one array of 63 points, each point representing a feature column in the final csv file.

4.2.2 Model Architectures:

The system is based on two deep learning models. The first model is the MediaPipe Hands landmark detection model, used to generate hand landmark coordinates. This model was used out of the box, without any modifications.

The second model was a custom trained TensorFlow Feed Forward Deep Neural Network to classify the gestures. This model was ultimately converted and deployed using the TFLite framework for faster inference and reduced model size. While the architecture is very simple, it proved effective for prototyping. This may also be attributed to the flattened nature of the input data.

Another plausible contributing factor to the system's excellent performance might be that the upstream model (MediaPipe Hands), was well trained on diverse datasets, and is able to output similarly distributed coordinates in various settings, once the coordinates were scaled. For the purposes of this project, we used the Min/Max scaler as presented in the Khazuhito00 [18] repository**.**

4.2.3 Model Training:

TensorFlow [17] was used to build and train the model. The model is a multilayer feedforward sequential network, consisting of dense layers and two dropout Layers. See **Figure 4.4** for the model architecture details.

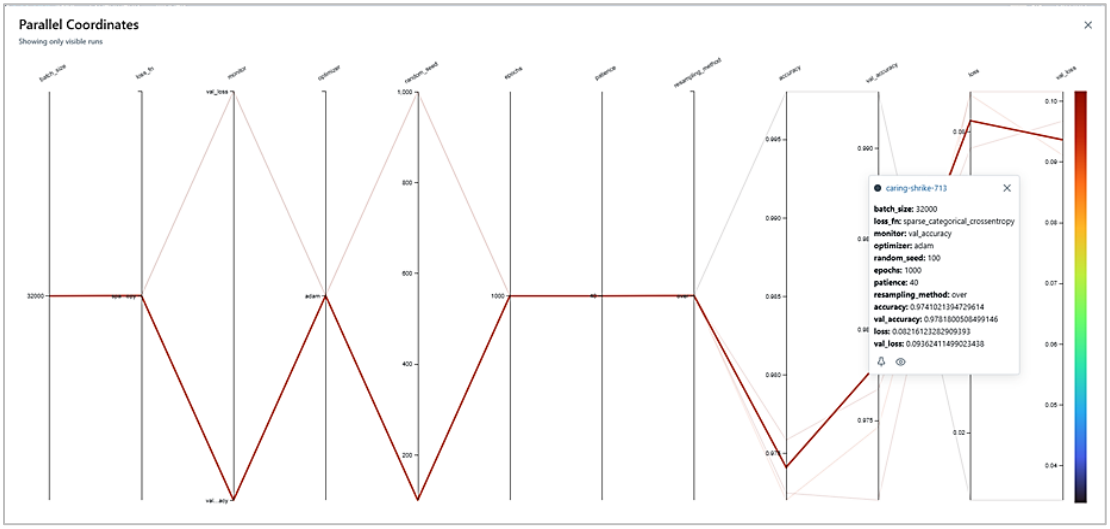


Figure 4.2: MLFlow Parallel Coordinates Chart. This figure shows the model performance for various hyperparameter experiments. The highlighted red line represents the final model. See **Figure 4.3** for more details.

The base model plateaued at about 96% accuracy and seemed to favor some classes over others as can be seen in the classification report in **Figure 4.5**.

It became apparent that the imbalance shown in the counts of classes in the initial exploratory data analysis (E.D.A.) came about when some of the classes were collected at higher rates than others. Over and under sampling of the classes was done using the Imbalanced-Learn library, with SMOTE [20] for oversampling and Near Miss [21] for under sampling to counteract these imbalances

4.2.4 Training Experimentation and Results:

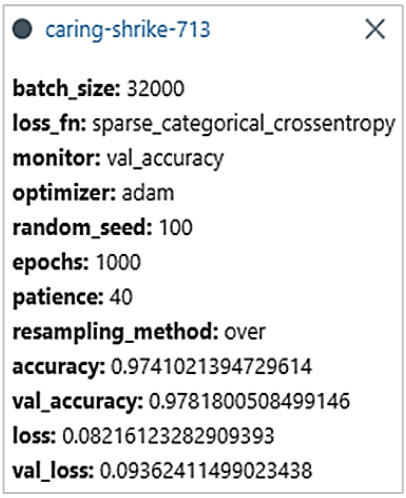


Figure 4.3 – Final Results and Hyperparameters. This figure shows the hyperparameter values and metrics for the final gesture classification model.

To find the best model, MLFlow [22] was used for experiment tracking. **Figures 4.2** to **4.6** show the training parameters used during experimentation and also provide insight into the architecture and performance of the model.

Figure 4.2 shows the parallel coordinates plot from MLFlow, while **Figure 4.3** shows a larger image of the parameters used to train the best performing model, which is highlighted by the red line in **Figure 4.2.** Although one other model exceeded the selected run, achieving above 99% Accuracy, the validation accuracy was much lower than the training accuracy, so that model was not selected due to potential overfitting.

4.2.5 Explanation of Training Hyperparameters:

Batch Size:

The Training Batch Size was constant at 32,000.

Loss Function:

The Loss Function was constant, utilizing Sparse Categorical Cross Entropy.

Monitor:

This was the parameter to monitor for early stopping. Validation Accuracy and Validation Loss were both used. The best performing model was selected using validation accuracy.

Optimizer:

The Adam Optimizer was used for all experiments.

Random Seed:

Random Seeds of 42, 100 and 1000 were used. These were used to determine the train-test split of the dataset, which was set at 75% train and 25% test using the SciKit-Learn library's train_test_split function. The shuffle parameter was set to the default value of True. All other parameters were left at the default value.

Epochs:

This hyperparameter represents the number of initial epochs. 1000 was used for all the experiments. None of them went above 369 epochs due to early stopping.

Patience:

Used for early stopping. This was the number of epochs after which the training should stop when there has been no improvement in the monitored metric. Patience values of 20 and 40 were used throughout experimentation.

Resampling Method:

The values of none, under and over were used, each corresponding to no resampling, under sampling or oversampling. The Imbalanced-Learn library was used for resampling, using **SMOTE** for oversampling and **Near Miss** for under sampling.

Accuracy, Validation Accuracy, Loss, Validation Loss:

These were the training metrics used to determine the quality of each model.

4.2.6 Model Details and Evaluation:

Figure 4.4 shows the architecture of the custom trained Deep Learning Feed Forward Network built using the TensorFlow framework. Each trainable layer, except the last, was a dense layer with a ReLU activation function. Two dropout layers were employed to reduce the chances of overfitting. The last layer employed a SoftMax activation to output the probabilities of each gesture.

Model: "sequential"

Layer (type)	Output Shape	Param #
dropout (Dropout)	(None, 63)	0
dense (Dense)	(None, 1400)	89,600
dropout_1 (Dropout)	(None, 1400)	0
dense_1 (Dense)	(None, 700)	980,700
dense_2 (Dense)	(None, 60)	42,060
dense_3 (Dense)	(None, 22)	1,342

Total params: 3,341,108 (12.75 MB)
Trainable params: 1,113,702 (4.25 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 2,227,406 (8.50 MB)

Figure 4.4: Model Architecture. This figure shows the architecture for the GCAI gesture classification model.

	precision	recall	f1-score	support
0	1.00	0.93	0.96	1630
1	1.00	0.92	0.96	1346
2	0.97	0.97	0.97	1153
3	1.00	0.99	0.99	1434
4	0.97	0.99	0.98	1031
5	0.96	1.00	0.98	1144
6	0.99	0.99	0.99	2735
7	0.99	0.99	0.99	1460
8	0.96	0.98	0.97	1470
9	0.95	0.99	0.97	1395
10	0.97	0.98	0.97	1464
11	0.99	0.99	0.99	1742
12	0.98	0.96	0.97	2290
13	0.99	0.97	0.98	1250
14	0.99	0.98	0.99	2314
15	0.99	0.98	0.98	1572
16	0.97	1.00	0.98	1638
17	0.94	0.98	0.96	2522
18	1.00	0.98	0.99	1536
19	0.99	0.99	0.99	1455
20	0.98	0.99	0.99	1390
21	0.99	0.99	0.99	1547
accuracy			0.98	35518
macro avg	0.98	0.98	0.98	35518
weighted avg	0.98	0.98	0.98	35518

Figure 4.5: Classification Report. This figure shows various statistics evaluating the performance of the model for each gesture class (left and right hand).

Figures 4.5 and 4.6 show the evaluation results of the model, after converting it to **TensorFlow Lite**, when evaluated using the test dataset. **Figure 4.5** shows a classification report demonstrating low variance in both precision and recall, while **Figure 4.6** shows a confusion matrix confirming that most of the predictions matched the ground truth labels.

Although the training loss slightly outpaced the validation loss, the difference was small enough to justify selection of this model. Furthermore, the validation accuracy outpaced the training accuracy, demonstrating minimal overfitting.

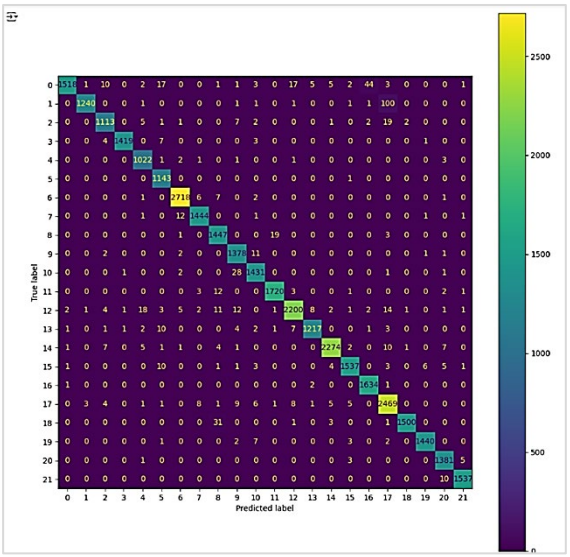


Figure 4.6: Confusion Matrix. This figure shows the model performance for classifying each gesture.

4.2.7 Training System:

Training was conducted using a Google Colab [30] Notebook, with Nvidia T4 and L4 GPU accelerators to speed up training. GPU acceleration was optional, due to the small size of the model, but reduced the training time by orders of magnitude. The System GPU RAM specifications were 15GB for the T4 and 22.5 GB for the L4, which are standard in a Google Colab environment and were much more than adequate for the size of the model.

NGROK [23] was integrated with the Colab notebook environment to provide the MLflow web user interface. This allowed for effortless logging of different iterations of training and experimentation with different hyperparameters. Overall, this setup was employed due to its flexibility and limited setup that proved optimal for model training.

4.3. Tools

4.3.1 Hardware

Raspberry Pi 5 (Model B, 8GB): Served as the main computing device, leveraging its compact design, processing power, and low energy consumption. It was chosen for its compatibility with edge AI applications and seamless integration with external peripherals. The Raspberry Pi managed the execution of models and supported real-time gesture recognition.

USB Camera: Used to capture video streams of hand gestures, providing the input for the computer vision pipeline. It interfaced with the Raspberry Pi to continuously capture video frames for gesture detection and classification.

4.3.2 Software

Linux OS: Utilized due to its compatibility with the Raspberry Pi hardware and the software packages used. Linux provided a stable and efficient environment for deploying the AI and gesture-recognition pipeline, ensuring robust system performance.

Python: The core programming language for developing the AI and backend systems. Python's rich ecosystem of libraries facilitated the implementation of gesture recognition models, preprocessing pipelines, and the integration of the frontend and backend systems.

MediaPipe: This framework was utilized for hand-tracking and extracting 3D landmark data from video streams. The pretrained MediaPipe Hands landmark detection model provided accurate and efficient coordinate outputs, which was a critical input for the custom classification model.

TensorFlow / TensorFlow Lite (TFLite): Used for building, training, and deploying the custom-trained neural network on the Raspberry Pi. TFLite optimized the original TensorFlow model for low-latency inference, ensuring real-time responsiveness.

OpenCV: Employed for image processing tasks, including capturing frames from the USB camera and preprocessing data before feeding it into the models. OpenCV enabled frame-by-frame manipulation to enhance the quality of inputs for gesture detection.

FastAPI: A lightweight backend framework used to create the RESTful API for communication between the frontend and the AI models. It ensured smooth data transfer and efficient endpoint management.

HTML/CSS and JavaScript: Utilized to build the web-based user interface, providing an intuitive platform for surgeons to interact with the system. JavaScript added dynamic elements to the interface, enhancing interactivity and responsiveness.

5. Results

5.1. Product Functionality

Core Components:

To explain the results, this section gives a brief technical overview. This system works by detecting the gestures of one hand as long as one hand is in view of the camera. Since the gesture detection model was trained on both right and left hands, either hand can be used. A total of eight gestures were employed for a combination of image panning and zooming capabilities.

The GCAI system supports both static medical images (x-rays, CT scans, MRIs, etc.) and live video feeds of medical imaging (intraoperative imaging). Regardless of the imaging type, the same gestures and image manipulations are available to the user.

5.1.1 Gesture Recognition Pipeline:

Zooming:

A Python function keeps track of the image/video's scale on the screen, using a preset factor to increase or reduce the scale. The current scaling factor is multiplied by a preset percentage and divided by 2 before being added to itself. This is done to smooth the zooming transition and reduce runaway increases that would otherwise occur from exponential growth at larger scales. This action is activated when the system detects the zooming gestures.

Panning (Index Finger):

A Python function keeps track of the image/video's (X, Y) position on the screen and adds/subtracts a preset number of pixels when the appropriate hand gesture is detected by the custom hand gesture model.

Panning (Palm):

This feature functions by translating the coordinates of the index finger that are outputted by the MediaPipe model into a position on the page. This is only active when the detected gesture is "palm". This is also tracked in the Python code, ultimately providing a real time hand tracking capability.

Freezing/Locking:

Useful for detailed examinations of the image, the two fingers gesture locks or unlocks onscreen transformations of the image. This works when the two fingers gesture is detected after any other gesture. A minimum number of continuous video frames with this gesture (configurable) must first be detected before this switch/toggle changes the status. If the status is "Locked", it becomes "Active" and vice-versa. A NumPy array keeps track of detections. A configurable inactivity lock activates the lock when no hand landmarks are detected after a preset inactivity duration. This is done to reduce unwanted changes to the image's scale/position due to unintentional detections.

5.1.2 Backend and Configurability

The backend, developed with **FastAPI**, ensures modularity and flexibility. Configurations are managed through environment variables, which flow through the system via class inheritance:

model_setup.py: Configures key parameters for model paths, confidence thresholds, and gesture labels.

inference.py: Implements the recognition pipeline, applying model configurations for live detection and classification.

main.py: Powers the FastAPI backend, integrating gesture recognition with real-time image manipulation endpoints.

These parameters include:

- Detection and tracking confidence thresholds.
- Camera input resolution.
- SSL support for secure data communication.
- File paths to models and directories.
- Image transformation.
- Inactivity and Locking.
- Session Management.
- Data Collection.

5.1.3 Frontend Interface

The web-based frontend, built with **HTML**, **CSS**, and **JavaScript**, provides a seamless interface for real-time gesture visualizations and system control. Features include:

- Live feedback on recognized gestures.
- Real-time display of manipulated medical images.
- Session storage and variables in the JavaScript code to keep track of the transformations.

Authentication and Data Management

To ensure secure and controlled access in medical environments, the **auth.py** module manages user authentication and session handling. Features include:

- **User Creation and Management**: Credential storage in a persistent **SQLite3** database.
- **Authentication**: Secure session validation to prevent unauthorized system access.

For non-networked operating environments, the users.db file can be periodically backed up. In networked setups, transitioning to robust storage and authentication options such as centralized hospital information systems (HIS) [24] or PACS-integrated [31] solutions is recommended.

Edge Deployment and Performance Optimization

The system is optimized for deployment in low-power environments. Key performance features include:

- **Quantized TFLite Models**: Reducing computational overhead while maintaining real-time gesture recognition.
- **Configurable Resolution and Parameters**: Adapting to varying hardware and camera setups to ensure optimal performance without compromising image clarity. While more research and testing are required, the initial prototype showed the potential for optimizations that can improve the stability of the application.

Security and Privacy Considerations

Maintaining data security is essential in surgical environments. The system includes:

- **SSL/TLS Encryption**: Enabling secure communication between devices. A self-signed certificate utility (create_certs.sh) simplifies setup for local deployments/development.
- **On-Device Data Storage**: Minimizing data exposure by storing user credentials and settings locally.

In most circumstances, we found that using the on-device deployment was adequate for this setup. However, for live imaging, more can be done to optimize the system for real time detection and real time video streaming from the imaging device.

5.2. Deployment Results

5.2.1 On Device Deployment

An initial attempt was made to deploy the program on the Raspberry Pi 4 Model B. However, a noticeable delay was consistently observed in running the MediaPipe landmark detection models in tandem with the gesture recognition neural networks given the memory and compute limitations. Transitioning to the Raspberry Pi 5 8 GB RAM yielded much improved performance, with no noticeable delay in the landmark detection or classification.

Additional experimentation was also conducted for further optimizing the latency of the AI models. Both the Coral USB Accelerator [35] and the Raspberry Pi AI HAT+ [38] were considered. The Coral USB Accelerator was ultimately chosen for experimentation due to its compatibility with the TFLite framework. Running the gesture classification neural network on the Coral USB Accelerator was successful, but this was ultimately not needed given the improved latency and performance using the Raspberry Pi 5.

5.2.2 Cloud Deployment:

Experiments were conducted to determine the viability of cloud deployment. While these proved useful, the application is currently an on-device deployment which can be moved to cloud or integrated with an appropriate cloud-based service as needed. The goals of the experiments outlined below were mainly to examine the latency for streaming video from the local environment to a cloud platform if larger, more sophisticated machine learning models were used or should on-device deployment be inadequate for real-time inference.

Experimentation was performed using 3 connection protocols, including **Websockets**, **RTSP** and **WebRTC**. WebRTC proved the most viable method as there was no noticeable delay in the video stream.

5.2.2.1 Experimental Findings on RTSP Connections with Google Cloud Vertex AI Vision

This section evaluates the feasibility of streaming video data from a Raspberry Pi to ***Google Cloud Vertex AI Vision*** [13] using the Real-Time Streaming Protocol (RTSP). The proposed architecture utilized the ***MediaMTX*** [32] Server as the RTSP server and the Google Cloud Platform (GCP) SDK for establishing a connection between the local Raspberry Pi environment and Google Cloud via the ***vaictl*** command-line tool. **Figure 5.1** shows a high-level overview of the proposed architecture. The implementation process involved authenticating the GCP SDK using the "gcloud init" command, setting up authentication on the MediaMTX server, installing FFmpeg, configuring the MediaMTX server to run at initialization, and sinking the webcam to the designated RTSP endpoint before executing the *vaictl* command.

Key Findings

Issues on Raspberry Pi:

While the RTSP connection from the Raspberry Pi to Google Cloud Vertex AI Vision was established, significant limitations in the Raspberry Pi architecture rendered direct streaming impractical without custom development. Specifically, the incompatibility of Google Cloud's client tools with the Raspberry Pi architecture necessitated a workaround. Successful testing was achieved by connecting via a Linux-based laptop using the MediaMTX server to relay video data to the cloud.

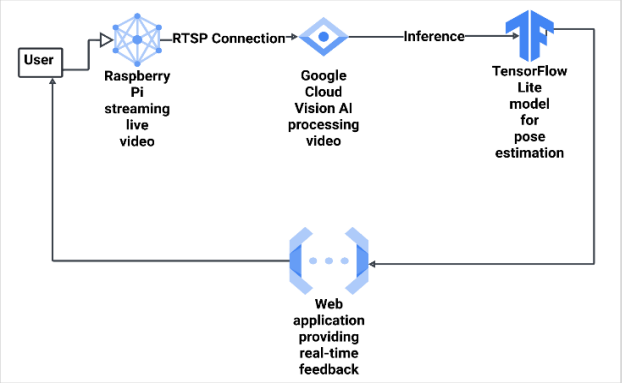


Figure 5.1: RTSP Connection Overview. This figure shows an overview of the RTSP architecture.

Noticeable Latency on Laptop:

When using a laptop to make the connection, significant delays (several seconds) were observed before video data was rendered in the cloud, even without AI processing. The latency appeared to stem from a combination of slow internet speeds and inefficiencies in RTSP communication. While faster internet connections may help mitigate this, further testing needs to be conducted to validate this assertion.

Potential Cloud-to-Cloud Solution:

A cloud-to-cloud streaming architecture could address these latency issues by removing the need for on-device processing. This approach would enable near-real-time feedback and lower response times during inference. Such a solution could also serve as a redundancy feature if on-device processing proves unreliable or inefficient. This

would involve running the inference application within the same cloud region as the Vertex AI workspace.

5.2.2.2 WebSockets Connection with JavaScript Client and High Latency Issues:

This section explores the use of WebSockets for streaming video data from a camera connected through a web browser, with a JavaScript-based client interfacing with a FastAPI server hosted on a ***Google Cloud Shell VM*** equipped with 16GB of memory. Initial testing focused on establishing a connection and evaluating latency performance before incorporating AI inference.

System Overview and Initial Testing:

WebSockets were implemented as the connection protocol, successfully enabling video streaming from the browser to the server. The system's initial configuration processed video data without AI inference, which was critical for assessing baseline latency. While the WebSockets setup demonstrated functional connectivity, significant latency was observed in the feedback loop, undermining the feasibility of real-time video streaming. This high latency persisted, leading to a decision to halt further testing, including inference evaluation, as the connection was deemed unsuitable for real-time AI tasks without further optimizations or hardware modifications.

Latency and Performance Observations:

High latency in the system prevented effective real-time video data retrieval. Despite initial functionality, the lag encountered during these tests highlighted the limitations of WebSockets in this context. Addressing these challenges may require protocol optimization or the adoption of alternative communication solutions better suited for low-latency applications.

Potential Cloud-to-Cloud Solution:

Similar to RTSP above, a solution that is fully hosted in a cloud environment might prove more viable.

Security Considerations

- **WebSockets Encryption**:

  The implementation of WSS (WebSockets over SSL) to encrypt data streams is necessary to safeguard transmitted video data. Ensuring secure connections is critical, particularly in cloud environments where high-latency streaming poses additional risks.

- **Data Privacy**:

  Compliance with data protection regulations is essential, necessitating strategies for secure authentication and authorization to prevent unauthorized access to the video stream.

Cost Considerations

- **Cloud Costs**:

  Running the FastAPI server in any cloud environment incurs expenses tied to resource utilization. Given the high latency without inference, identifying cost-effective alternatives or optimizing the existing setup for faster processing is imperative.

- **Bandwidth Costs**

  High-latency connections often result in increased bandwidth consumption. Techniques such as data compression and frame rate adjustments may reduce transmission overhead, thereby lowering associated costs. However, the costs of decoding the compressed data might result in added latency and should be considered equally.

5.2.2.3 WebRTC Connection with JavaScript Client and AI Inference on Google Cloud

This section evaluates the implementation of WebRTC for real-time video streaming and AI inference using a JavaScript-based client and an ***aiortc*** server hosted on Google Cloud. The system is designed to process video streams captured through a web browser, with low-latency performance as the primary goal. **Figure 5.2** shows a high-level overview of the proposed WebRTC architecture.

System Overview and Initial Testing

WebRTC was utilized as the communication protocol, leveraging ***Google's free STUN servers*** to achieve near-real-time performance. The server, hosted on a Google Cloud Shell VM with 16GB of RAM, efficiently handled single-hand gesture inference with minimal latency. Plans for production involve transitioning the system to Google Cloud Run Instances, enabling scalability and operational efficiency.

Inference Performance and Latency Observations

The system employed the **MediaPipe Hand Landmark Detection Model** with a setup based on the Khazuhito00 repository. Tests were conducted at two input resolutions to

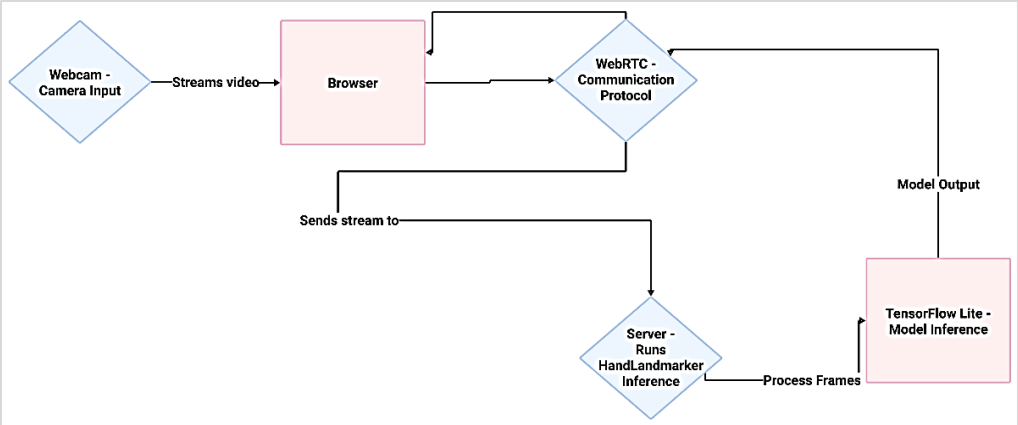


Figure 5.2: WebRTC Connection Overview. This figure provides a high-level overview of the WebRTC architecture.

evaluate latency: 160x120 and 320x240. While single-hand inference achieved low latency without the need for GPUs, significant latency spikes were observed during two-hand inference. Interestingly, resolution changes had no measurable impact on these delays, suggesting a computational bottleneck in multi-hand processing.

To address this, possible solutions include:

- **GPU Acceleration**: Leveraging GPUs to reduce computation time for multi-hand inference.
- **Code Optimization**: Investigating the inference pipeline for potential inefficiencies.

Further investigation is needed to pinpoint the root cause of the latency with two-hand inputs.

Security Considerations

- **WebRTC Encryption**:

  Implementing secure protocols, such as **DTLS *(Datagram Transport Layer Security)*** [33], is essential to encrypt WebRTC data streams and safeguard user data.
- **Data Privacy**:

  Ensuring compliance with data protection regulations and employing robust authentication mechanisms to prevent unauthorized access are critical for maintaining data integrity and user trust.
- **Cloud Security**:

  Cloud-based deployments must adhere to security best practices, including regular monitoring and securing sensitive APIs to prevent breaches.

Cost Considerations

- **Cloud Costs**:

  Moving the system to Google Cloud Run Instances will incur costs based on resource utilization, including compute time and data transfer. Cost-effective scaling options are needed to handle high traffic without incurring excessive expenses.
- **GPU vs. CPU Usage**:

  While single-hand inference does not necessitate GPUs, their use for multi-hand inference may become essential. A cost-benefit analysis will determine whether GPU-enabled instances are justified for achieving low latency in two-hand gesture recognition.
- **Bandwidth Costs**:

  High-resolution streaming could increase data transfer expenses on Google Cloud. Optimizing for lower bandwidth usage, such as adaptive bitrate streaming or resolution scaling, can reduce these costs, particularly for multi-hand inputs.

By leveraging WebRTC's low-latency capabilities and optimizing inference performance, this system demonstrates potential for scalable and efficient AI-driven gesture recognition. The outlined next steps aim to address current challenges, ensuring robust real-time functionality for both single- and multi-hand input scenarios.

5.3. Challenges

5.3.1 Detection for Colored Gloves

A primary challenge encountered during the project was ensuring accurate detection of blue gloves in the video feed. Initial attempts using standard color thresholds in the RGB and HSV color spaces led to inconsistent results due to varying lighting conditions and shadows, which caused missed detections.

To address this, we experimented with different color space conversions in OpenCV to identify a more robust method for segmenting the glove color from the background. After extensive testing, the BGR2LUV color space conversion was found to provide the most consistent results for detecting blue gloves.

The LUV color space demonstrated resilience against changes in lighting and ensured reliable key point detection, thereby improving the accuracy of gesture classification. The experimentation focused exclusively on blue gloves, and the methodology may need adaptation for other glove colors in future applications.

For clear and white gloves, the default setting for BGR2RGB color conversion worked well. Further experimentation will be necessary to evaluate GCAI's performance with more glove colors.

5.3.2 Optimizing Streaming from a Second Camera

Integrating a second camera for simultaneous video streaming to simulate a live intraoperative imaging feed posed another challenge. While the system successfully captured video streams, a slight delay was observed between the source video feed and the output displayed on the screen. This latency affected the responsiveness of the application, particularly for real-time gesture recognition tasks. Although the issue was not fully resolved during this phase of the project, it was identified as a priority for future work. Potential strategies for addressing this delay include optimizing video frame buffering, reducing the resolution of the streamed video, or leveraging hardware-accelerated video processing.

6. Next Steps

Future development will focus on enhancing system versatility and usability. Other potential additions include integrating multimodal interaction options, such as voice activation and foot pedals, and transitioning to a cloud-edge hybrid deployment for greater scalability and reliability. Advanced user-specific adaptations will improve accuracy, while user tracking will mitigate interference for other individuals captured by the video. Additionally, expanding capabilities to handle more complex medical imaging and video data types will help further increase GCAI's role in modern surgical and medical imaging practices.

7. Conclusion

The Gesture-Controlled AI Assistant (GCAI) showcases a promising step forward in improving medical image navigation during intraoperative imaging. By leveraging contactless gesture controls, it enhances surgical precision and workflow efficiency while maintaining sterility. Initial deployment using the TFLite framework on the Raspberry Pi 5 highlights its practicality, versatility, and security. So far, GCAI has demonstrated strong performance in static image manipulation to detect blue, clear, and white gloves. Areas such as live video streaming and specialized color space conversions present opportunities for refinement, ensuring the system can meet the dynamic demands of surgical environments.

8. References:

[1] *M.2 HAT+ - Raspberry Pi Documentation*. (2024). Raspberrypi.com. https://www.raspberrypi.com/documentation/accessories/m2-hat-plus.html

[2] Moody, T. (2023, December 8). *Transform Your WebCam into an IP Camera (for Frigate)*. Medium. https://medium.com/@timothydmoody/transform-your-webcam-into-an-ip-camera-for-frigate-8cf50fd749e9

[3] Mukherjee, K. (2021, May 19). *Convert any Webcam into IPCam in 2 minutes and Stream FullHD(1920*1080) Videos over Internet at 35 fps Using FFMPEG Powered By CUDA*. Medium. https://kaustavmukherjee-66179.medium.com/convert-any-webcam-into-ipcam-in-2-minutes-using-rtsp-server-and-ffmpeg-a27414f08af5

[4] *hailo-ai. (2024). hailo-rpi5-examples/doc/install-raspberry-pi5.md at main · hailo-ai/hailo-rpi5-examples. GitHub. https://github.com/hailo-ai/hailo-rpi5-examples/blob/main/doc/install-raspberry-pi5.md#how-to-set-up-raspberry-pi-5-and-hailo-8l*

[5] hailo-ai. (2024a). *hailo-rpi5-examples/doc/basic-pipelines.md at main · hailo-ai/hailo-rpi5-examples*. GitHub. https://github.com/hailo-ai/hailo-rpi5-examples/blob/main/doc/basic-pipelines.md#running-with-different-input-sources

[6] Dwivedi, H. (2023, July 19). *Running TensorFlow Lite Image Classification Models in Python*. Comet. https://www.comet.com/site/blog/running-tensorflow-lite-image-classification-models-in-python/

[7] *sklearn.metrics.classification_report — scikit-learn 0.20.3 documentation*. (2018). Scikit-Learn.org. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html

[8] *sklearn.metrics.ConfusionMatrixDisplay*. (n.d.). Scikit-Learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html

[9] *LiteRT Model Analyzer*. (2024). Google AI for Developers. https://ai.google.dev/edge/litert/models/model_analyzer

[10] *FastAPI*. (n.d.). Fastapi.tiangolo.com. https://fastapi.tiangolo.com/

[11] *Starlette*. (n.d.). Www.starlette.io. https://www.starlette.io/

[12] *Create streams and ingest data*. (2024). Google Cloud. https://cloud.google.com/vision-ai/docs/create-manage-streams

[13] *Vertex AI Vision*. (n.d.). Google Cloud. https://cloud.google.com/vertex-ai-vision

[14] MDN Contributors. (2023, September 25). *JavaScript*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/javascript

[15] *Python*. (2019, May 29). Python.org; Python.org. https://www.python.org/

[16] W3Schools. (2024). *W3Schools Online Web Tutorials*. W3schools.com; W3Schools. https://www.w3schools.com/

[17] *TensorFlow*. (2019). TensorFlow; Google. https://www.tensorflow.org/

[18] Takahashi, S. (2020, December 1). *hand-gesture-recognition-using-mediapipe*. GitHub. https://github.com/Kazuhito00/hand-gesture-recognition-using-mediapipe

[19] *MediaPipe Solutions guide | Edge*. (n.d.). Google for Developers. https://ai.google.dev/edge/mediapipe/solutions/guide

[20] *SMOTE — Version 0.9.0*. (n.d.). Imbalanced-Learn.org. https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

[21] *NearMiss — Version 0.12.4*. (2014). Imbalanced-Learn.org. https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.NearMiss.html

[22] *MLflow - A platform for the machine learning lifecycle*. (n.d.). MLflow. https://mlflow.org/

[23] *ngrok - secure introspectable tunnels to localhost*. (2019). Ngrok.com. https://ngrok.com/

[24] Talking HealthTech. (2020, October 19). *What is Hospital Information Systems (HIS)?* Talking HealthTech. https://www.talkinghealthtech.com/glossary/hospital-information-systems-his

[25] *Sterile Processing and the Operating Room: Why Patient Safety Can't Be Rushed · Incision*. (2023, October 4). Incision. https://www.incision.care/blog/sterile-processing-and-or-why-patient-safety-cannot-be-rushed

[26] *Intraoperative Imaging Market Size And Share Report, 2030*. (2023). Grandviewresearch.com. https://www.grandviewresearch.com/industry-analysis/intraoperative-imaging-market

[27] Hughes, A. (2012, June 7). *Kinect Launches a Surgical Revolution*. Microsoft Research. https://www.microsoft.com/en-us/research/blog/kinect-launches-surgical-revolution/

[28] *GestSure*. (2024). https://www.gestsure.com

[29] Mentis, H. M., O'Hara, K., Gonzalez, G., Sellen, A., Corish, R., Criminisi, A., Trivedi, R., & Theodore, P. (2015). Voice or Gesture in the Operating Room. *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. https://doi.org/10.1145/2702613.2702963

[30] *Google Colaboratory*. (2019). Google.com. https://colab.research.google.com/

[31] *Radiology image management PACS*. (n.d.). Philips. https://www.usa.philips.com/healthcare/product/HCNOCTN424PACS/diagnostic-workspace?utm_id=71700000120532603&origin=7_700000002938335_71700000120532603_58700008816121760_43700081144782673&gad_source=1&gclid=CjwKCAiA6t-6BhA3EiwAltRFGEnvaf7zKsdSxALwdLs6Vc5dBXpvRjS

__NFA1rykIj50L_WLUxuyRRoCIscQAvD_BwE&gclsrc=aw.ds

[32] blueenviron. (2024, March 24). *MediaMTX*. GitHub. https://github.com/bluenviron/mediamtx

[33] *DTLS (Datagram Transport Layer Security) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. (2023, June 8). Developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Glossary/DTLS

[34] *How to Make the Hand Detection of Mediapipe Work on Hand with Gloves.* (2023, May 24). Stack Overflow. https://stackoverflow.com/questions/76325975/how-to-make-the-hand-detection-of-mediapipe-work-on-hand-with-gloves

[35] *Coral*. (n.d.). Coral. https://coral.ai/

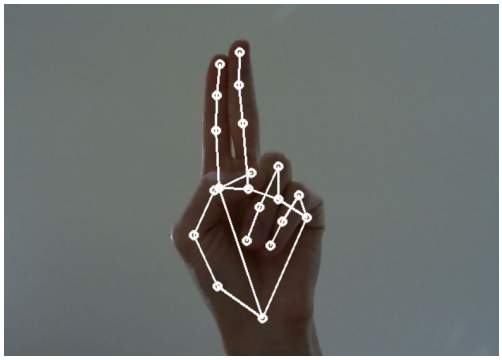
[36] *Run inference on the Edge TPU with Python | Coral*. (2020). Coral. https://coral.ai/docs/edgetpu/tflite-python/#update-existing-tf-lite-code-for-the-edge-tpu

[37] Rincker, R. (2014, July 9). *Vision Standards Make the Cut for Medical Imaging*. Medicaldesignbriefs.com. https://www.medicaldesignbriefs.com/component/content/article/20075-vision-standards-make-the-cut-for-medical-imaging

[38] *Raspberry Pi AI HAT+ – Raspberry Pi*. (2024). Raspberry Pi. https://www.raspberrypi.com/products/ai-hat/

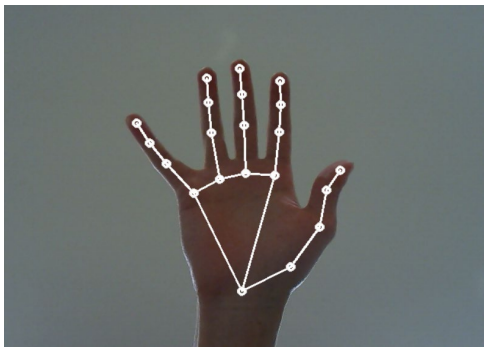
9. Appendix A: Example Gestures

Gesture 1: Lock / Unlock Image



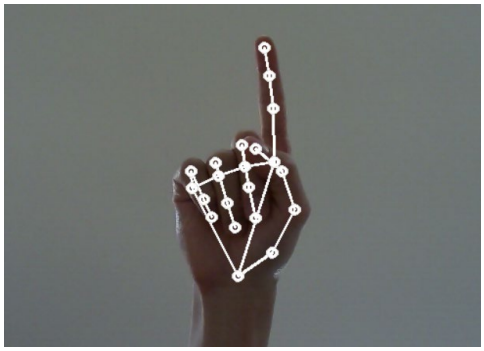
This gesture freezes or unfreezes the image position.

Gesture 2: Free-Range Panning



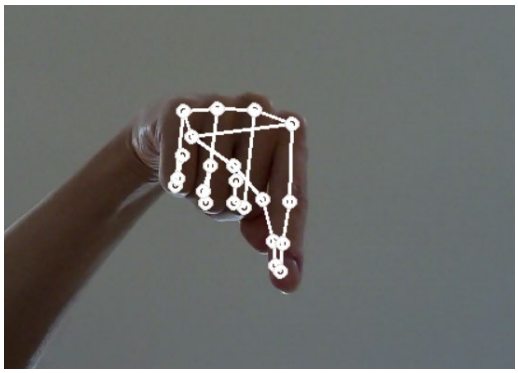
With this gesture, the image moves to match with the hand's motion (hand tracking).

Gestures 3: Directional Panning - Up



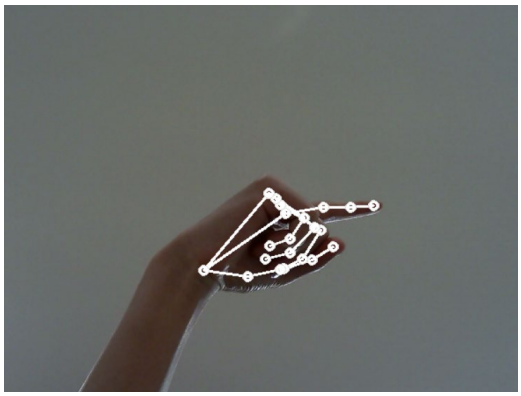
This gesture moves the image upwards.

Gestures 4: Directional Panning – Down



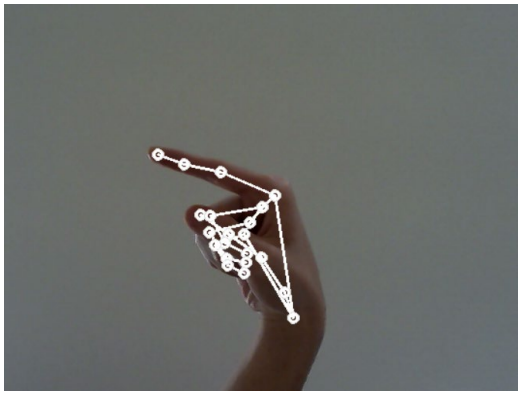
The gesture moves the image downwards.

Gestures 5: Directional Panning - Right



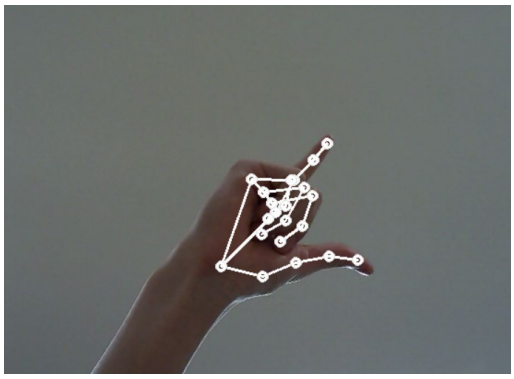
The gesture moves the image to the right.

Gestures 6: Directional Panning – Left



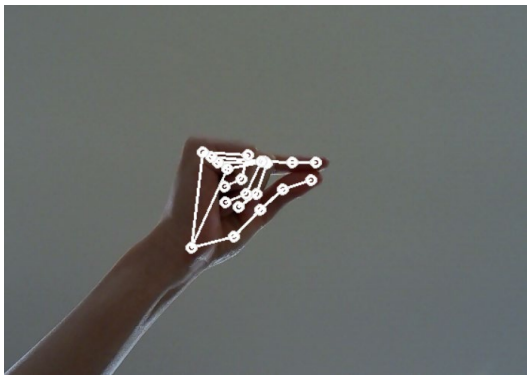
The gesture moves the image to the left.

Gesture 7: Zoom In



This gesture allows the user to zoom in on the image.

Gesture 8: Zoom Out



This gesture allows the user to zoom out of the image.