Real-Time Systems Course Project Report

Virtual Eyeglass Try-On System Using ESP32-S3 and Computer Vision

Course: Real-Time Systems
Student: Giga Shubitidze
Teacher: Saulius Sakavičius
Group: DISfmu-24

Contents

Abstract

This report documents the development of a real-time virtual eyeglass try-on system that uses the ESP32-S3 microcontroller with integrated camera module, computer vision techniques, and generative AI. The system streams video from an ESP32-S3 camera module over WiFi and processes it in real-time using facial detection algorithms to overlay virtual eyeglasses onto the user's face. What distinguishes this implementation is the integration with my master's thesis research on Generative Adversarial Networks (GANs) for creating synthetic eyeglass frame images, thus addressing the shortage of diverse eyewear datasets. The system successfully demonstrates practical applications of embedded systems programming, real-time video processing, computer vision, and machine learning in creating an interactive augmented reality experience.

1. Introduction

1.1 Project Overview

The real-time virtual eyeglass try-on system combines hardware and software components to create a practical application for the eyewear industry. Users can virtually "try on" different eyeglass frames in real-time through a camera feed, eliminating the need for physical trial and error when selecting eyewear. This technology offers significant advantages for both consumers and retailers in the eyewear market.

The complete source code for this project is available on GitHub:

https://github.com/bokuwagiga/virtual_eyeglasses_tryon

1.2 Motivation

The motivation behind this project comes from several factors:

- **Consumer Convenience**: Traditional eyeglass purchasing requires physically trying multiple frames, which is time-consuming and often restricted by in-store inventory.
- **COVID-19 Impact**: The pandemic accelerated the need for contactless solutions in retail settings.
- **Integration with Research**: The opportunity to apply my master's thesis research on generative AI for eyeglass frame images to a practical, real-world implementation.
- **Technical Challenge**: Combining embedded systems, networking, computer vision, and machine learning in a single real-time application represents a comprehensive technical challenge.
- **Personal Experience**: As a regular eyeglass user, I have often found the traditional process of selecting frames inconvenient. A virtual try-on system would significantly improve the experience by providing a preliminary sense of which styles best suit my face before visiting a store.

1.3 Project Objectives

The primary objectives of this project were to:

1. Create a stable real-time video streaming solution using the ESP32-S3 microcontroller
2. Develop accurate facial detection and landmark tracking algorithms for proper eyeglass positioning
3. Design a user-friendly interface for eyeglass selection and visualization
4. Integrate GAN-generated eyeglass frames from my master's thesis research
5. Ensure the system operates with minimal latency to maintain a responsive user experience

2. Background and Related Work

2.1 Virtual Try-On Technologies

Virtual try-on technologies have gained significant traction in recent years, particularly in fashion and cosmetics industries. These technologies often employ augmented reality (AR) to overlay virtual products onto real-world images or video feeds. In the context of eyewear, several commercial solutions exist, including those offered by major retailers like Warby Parker and EyeBuyDirect. However, most of these solutions rely on powerful smartphones or web applications rather than dedicated embedded systems.

2.2 ESP32-S3 Platform

The ESP32-S3 is a dual-core microcontroller from Espressif Systems featuring integrated Wi-Fi capabilities. The specific model used in this project includes a camera interface, making it suitable for computer vision applications. The ESP32-S3's combination of processing power, wireless connectivity, and low power consumption makes it an ideal platform for edge computing applications like this project.

2.3 Computer Vision for Facial Analysis

Face detection and facial landmark detection are fundamental computer vision tasks that have seen significant improvements through the use of machine learning. The Haar Cascade classifier, developed by Viola and Jones, provides an efficient method for face detection, while the Local Binary Features (LBF) model offers accurate facial landmark detection. These technologies are crucial for correctly positioning virtual eyeglasses on a user's face.

2.4 Generative Adversarial Networks (GANs)

GANs, introduced by Goodfellow et al. in 2014, consist of two neural networks—a generator and a discriminator—that compete against each other to create realistic synthetic data. As part of my master's thesis, I applied GAN architectures to generate diverse eyeglass frame images,

addressing the challenge of limited dataset availability in the eyewear domain. This project utilizes these generated images to expand the variety of frames users can virtually try on.

3. System Architecture

3.1 Hardware Configuration

The system hardware centers around the ESP32-S3 development board with an integrated camera module. The setup includes:

- ESP32-S3 development board
- OV2640 camera module
- Wi-Fi router for local network connectivity
- Computer for running the Python-based user interface

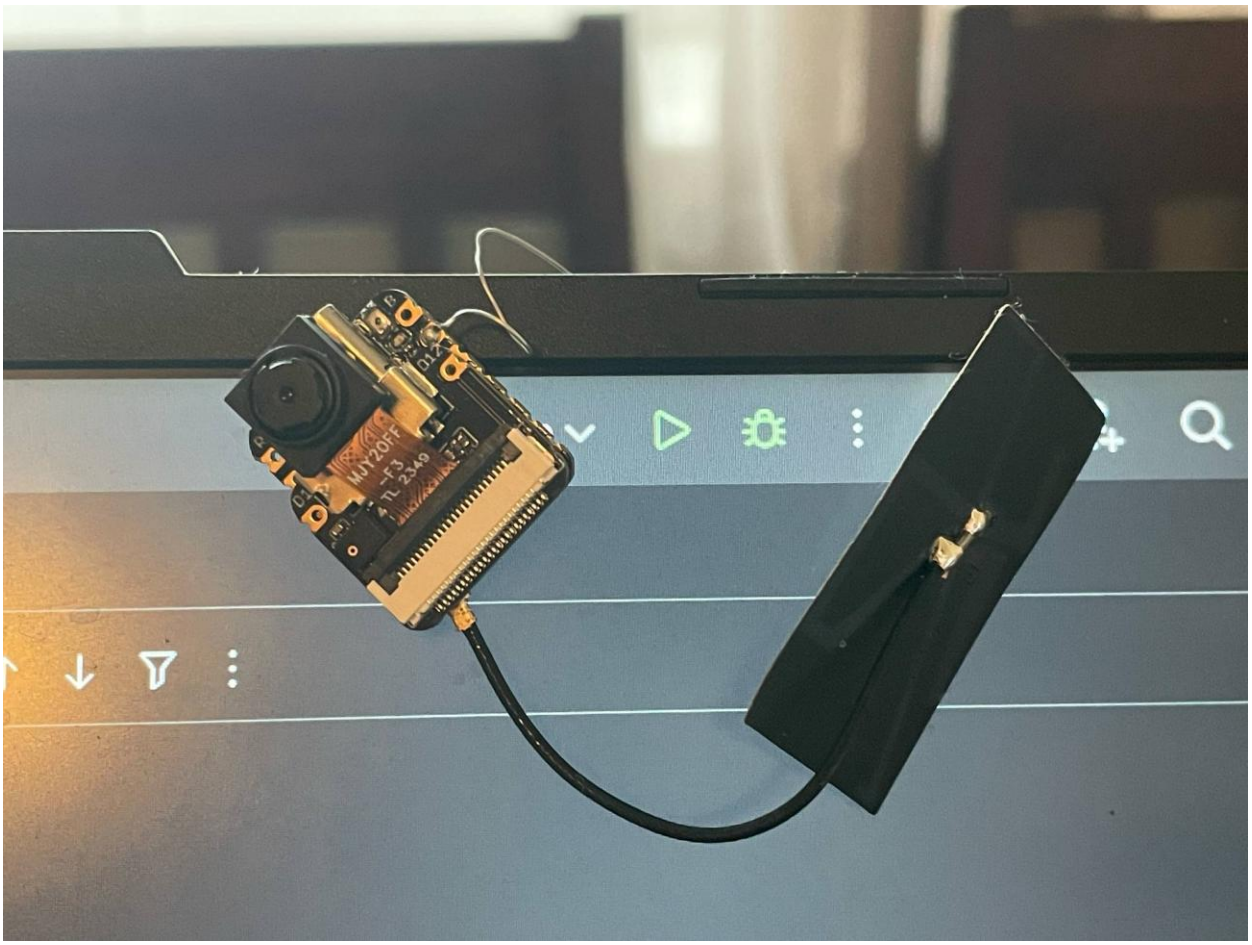

Figure 1 ESP32-S3 hardware setup with camera module

3.2 Software Components

The software architecture consists of two main components:

1. **ESP32-S3 Firmware**: Written in C using the ESP-IDF framework, responsible for:
    - Camera initialization and configuration
    - Image capture
    - MJPEG encoding
    - HTTP server implementation for video streaming
2. **Python Application**: Developed using OpenCV, Tkinter, and other libraries, responsible for:
    - Connecting to the ESP32-S3 video stream
    - Face detection and facial landmark tracking
    - Virtual eyeglass overlay processing
    - User interface for eyeglass selection

3.3 Communication Protocol

Communication between the ESP32-S3 and the Python application occurs over HTTP using an MJPEG (Motion JPEG) stream. The ESP32-S3 acts as an HTTP server, continuously capturing and encoding frames from the camera, which are then sent as a multipart response to the client application. This approach offers several advantages:

- **Compatibility**: HTTP is widely supported and easily implemented
- **Simplicity**: No need for complex streaming protocols like RTSP
- **Flexibility**: Easy to integrate with web browsers and various programming languages
- **Efficiency**: MJPEG provides reasonable compression while maintaining image quality

3.4 Data Flow

The system's data flow follows these steps:

1. The ESP32-S3 captures camera frames at approximately 30 FPS
2. Frames are JPEG-encoded and sent over Wi-Fi via an HTTP MJPEG stream
3. The Python application receives and decodes the JPEG frames
4. Face detection algorithms locate faces in the frame
5. If available, facial landmark detection identifies key facial points
6. Virtual eyeglasses are resized, rotated, and positioned based on facial geometry
7. The augmented frame is displayed in the user interface
8. The process repeats for each new frame

4. Implementation Details

4.1 ESP32-S3 Firmware Implementation

The ESP32-S3 firmware was developed using the Espressif IoT Development Framework (ESP-IDF), which provides a comprehensive set of libraries and tools for ESP32 series

microcontrollers. The implementation focuses on efficient camera operation and robust HTTP streaming.

4.1.1 Camera Configuration

The camera is configured with specific pins for the SCCB interface and data lines, optimized for the ESP32-S3 hardware:

```
camera_config_t config = {
    .pin_pwdn = -1,
    .pin_reset = -1,
    .pin_xclk = 10,
    .pin_sccb_sda = 40,
    .pin_sccb_scl = 39,

    .pin_d7 = 48,
    .pin_d6 = 11,
    .pin_d5 = 12,
    .pin_d4 = 14,
    .pin_d3 = 16,
    .pin_d2 = 18,
    .pin_d1 = 17,
    .pin_d0 = 15,

    .pin_vsync = 38,
    .pin_href = 47,
    .pin_pclk = 13,

    .xclk_freq_hz = 20000000,
    .ledc_timer = LEDC_TIMER_0,
    .ledc_channel = LEDC_CHANNEL_0,

    .pixel_format = PIXFORMAT_JPEG,
    .frame_size = FRAMESIZE_QVGA,
    .jpeg_quality = 12,
    .fb_count = 3,
    .grab_mode = CAMERA_GRAB_LATEST,
    .fb_location = CAMERA_FB_IN_PSRAM
};
```

Figure 2 ESP32S3 Camera Configuration

The configuration uses QVGA resolution (320x240) to balance image quality and performance, with JPEG encoding performed by the camera hardware to reduce CPU load.

4.1.2 HTTP Server and MJPEG Streaming

The HTTP server uses the ESP-IDF httpd component to handle incoming connections and serve the MJPEG stream:

```
esp_err_t stream_handler(httpd_req_t *req) {
    camera_fb_t *fb = NULL;
    esp_err_t res = ESP_OK;

    res = httpd_resp_set_type(req, "multipart/x-mixed-replace; boundary=frame");
    if (res != ESP_OK) return res;

    while (true) {
```

```
        fb = esp_camera_fb_get();
        if (!fb) {
            ESP_LOGE(TAG, "Camera capture failed");
            continue;
        }

        char part_buf[64];
        snprintf(part_buf, sizeof(part_buf),
                 "--frame\r\nContent-Type: image/jpeg\r\nContent-Length: %u\r\n\r\n",
                 fb->len);

        res = httpd_resp_send_chunk(req, part_buf, strlen(part_buf));
        res |= httpd_resp_send_chunk(req, (char *)fb->buf, fb->len);
        res |= httpd_resp_send_chunk(req, "\r\n", 2);

        esp_camera_fb_return(fb);
        if (res != ESP_OK) break;

        vTaskDelay(30 / portTICK_PERIOD_MS); // ~30 fps
    }

    return res;
}
```

Figure 3 Handling MJPEG Streaming

The `stream_handler` function continuously captures frames, formats them according to the MJPEG standard, and sends them to the client. The `vTaskDelay` function ensures that the system doesn't capture frames faster than necessary, maintaining approximately 30 FPS.

4.1.3 Wi-Fi Connectivity

The firmware establishes a Wi-Fi connection as a station (client) to an existing network:

```
void wifi_init_sta() {
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = WIFI_SSID,
            .password = WIFI_PASS,
            .threshold.authmode = WIFI_AUTH_WPA2_PSK,
        },
    };

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
    ESP_ERROR_CHECK(esp_wifi_start());

    // ... connection and IP display code ...
}
```

Figure 4 Wi-Fi initialization

The system logs its IP address to the serial console once connected, allowing the user to access the stream URL.

```
I (1196) wifi:dp: 1, bi: 102400, li: 3, scale listen interval from 307200 us to 307200 us
I (1206) wifi:set rx beacon pti, rx_bcn_pti: 0, bcn_timeout: 25000, mt_pti: 0, mt_time: 10000
I (1276) wifi:AP's beacon interval = 102400 us, DTIM period = 1
I (1556) cam_stream: Connected to SSID:TP-Link_5CD9
I (1556) cam_stream: Camera Ready! Stream: http://0.0.0.0/mjpeg
I (1556) s3 ll_cam: DMA Channel=1
I (1556) cam_hal: cam init ok
I (1556) sccb-ng: pin_sda 40 pin_scl 39
I (1556) sccb-ng: sccb_i2c_port=1
I (1576) camera: Detected camera at address=0x30
I (1576) camera: Detected OV2640 camera
I (1576) camera: Camera PID=0x26 VER=0x42 MIDL=0x7f MIDH=0xa2
I (1646) cam_hal: buffer_size: 16384, half_buffer_size: 1024, node_buffer_size: 1024, node_cnt: 16, total_cnt: 15
I (1646) cam_hal: Allocating 15360 Byte frame buffer in PSRAM
I (1656) cam_hal: Allocating 15360 Byte frame buffer in PSRAM
I (1656) cam_hal: Allocating 15360 Byte frame buffer in PSRAM
I (1666) cam_hal: cam config ok
I (1666) ov2640: Set PLL: clk_2x: 0, clk_div: 0, pclk_auto: 0, pclk_div: 8
I (1736) ov2640: Set PLL: clk_2x: 0, clk_div: 0, pclk_auto: 0, pclk_div: 8
W (1736) cam_hal: NO-SOI
W (1746) cam_hal: NO-SOI
I (1776) main_task: Returned from app_main()
I (2906) esp_netif_handlers: sta ip: 192.168.0.102, mask: 255.255.255.0, gw: 192.168.0.1
```

Figure 5 ESP32-S3 Serial output

4.2 Python Application Implementation

The Python application serves as the client for the ESP32-S3's video stream and implements the virtual try-on functionality. It was developed using OpenCV for computer vision tasks and Tkinter for the graphical user interface.

4.2.1 Video Stream Processing

The application connects to the ESP32-S3's MJPEG stream and processes the incoming frames:

```python
try:
    stream = urllib.request.urlopen(stream_url, timeout=10)
    print("Connected to stream successfully")
except Exception as e:
    print(f"Failed to connect to stream: {e}")
    # Error handling...

# Get the multipart boundary string from headers
headers = stream.info()
content_type = headers.get('Content-Type', '')
boundary_match = re.search(r'boundary=(.*?)(?:$|;|\s)', content_type)
if boundary_match:
    boundary = boundary_match.group(1)
else:
    boundary = "frame"  # Default boundary
```

Figure 6 Snippet from Parsing MJPEG Stream

The application parses the MJPEG stream by finding boundary markers and extracting individual JPEG frames, which are then decoded using OpenCV.

4.2.2 Face Detection and Landmark Tracking

Face detection uses OpenCV's Haar Cascade classifier to locate faces in each frame:

```
# Convert frame to grayscale for face detection
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# Detect faces with more lenient parameters
faces = face_cascade.detectMultiScale(
    gray,
    scaleFactor=1.05,
    minNeighbors=3,
    minSize=(30, 30)
)
```

Figure 7 Face Detection using Haar Cascade classifier

When a face is detected, the application attempts to identify facial landmarks using the Local Binary Features (LBF) model:

```
if facemark is not None:
    # Convert detected faces to OpenCV format
    faces_for_landmarks = []
    for (x, y, w, h) in faces:
        faces_for_landmarks.append([x, y, w, h])

    ok, landmarks = facemark.fit(gray, np.array(faces_for_landmarks))
    if ok and len(landmarks) > 0:
        face_landmarks = landmarks
        main.last_landmarks = face_landmarks
```

Figure 8 Identifying facial landmarks using LBF model

These landmarks are used to accurately position the virtual eyeglasses.

4.2.3 Eyeglass Overlay Implementation

The application overlays virtual eyeglasses onto the detected face using alpha blending:

```
def overlay_glasses(frame, face, glasses_img, landmarks=None):
    # ... positioning code ...

    # For each color channel, blend the image
    for c in range(0, 3):
        # Get ROI for this operation
        roi = frame[roi_y:roi_y + roi_h, roi_x:roi_x + roi_w, c]

        # Get corresponding part of glasses and alpha
        glasses_roi = resized_glasses[:roi_h, :roi_w, c]
        alpha_roi = alpha_mask[:roi_h, :roi_w]

        # Apply blending
        frame[roi_y:roi_y + roi_h, roi_x:roi_x + roi_w, c] = (
                roi * (1 - alpha_roi) + glasses_roi * alpha_roi
        )
```

```
    return frame
```
Figure 9 Overlaying virtual eyeglasses function

To enhance realism, the system:

- Rotates eyeglasses to match the angle of the eyes
- Scales eyeglasses based on interpupillary distance
- Applies motion smoothing to prevent jitter
- Handles transparent backgrounds in eyeglass images

4.2.4 User Interface Design

The application's user interface was built using Tkinter, featuring:

- A video display area showing the live camera feed with eyeglass overlay
- A scrollable gallery of eyeglass options with pagination
- Navigation buttons for browsing through available eyeglasses

```
def create_ui(glasses_dir):
    # Create main Tkinter window
    glasses_window = Tk()
    glasses_window.title("Virtual Eyewear Try-on")
    glasses_window.geometry("800x600")

    # Create frame for video display (top portion)
    video_frame = Frame(glasses_window, bg="black")
    video_frame.pack(side="top", fill="both", expand=True)

    # Create label for video display
    video_label = Label(video_frame)
    video_label.pack(pady=10)

    # ... scrolling gallery implementation ...

    return glasses_window, video_label
```
Figure 10 Function to Implement User-Interface

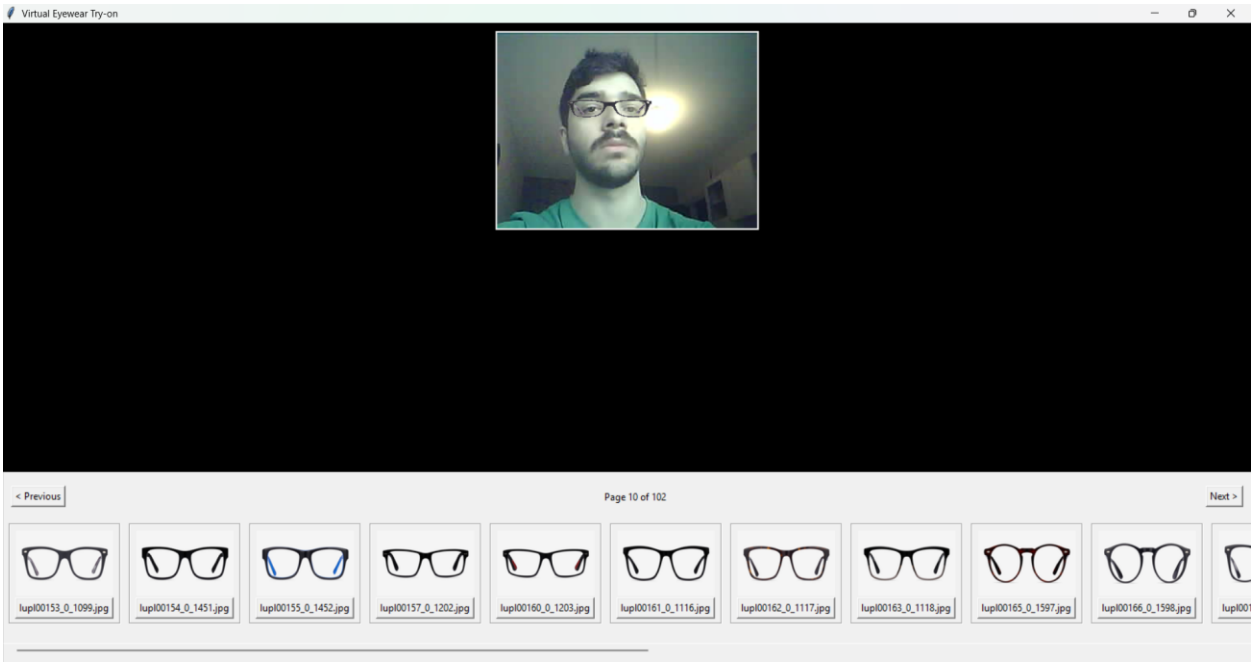

Figure 11 Screenshot of the system user interface

4.3 Integration with GAN-Generated Eyeglass Images

A unique aspect of this project is the integration with my master's thesis research on Generative Adversarial Networks for eyeglass frame image generation.

4.3.1 GAN Architecture Overview

For my thesis, I implemented a Deep Convolutional GAN (DCGAN) architecture to generate synthetic eyeglass frames. The DCGAN consists of:

- A generator network that creates synthetic eyeglass images from random noise
- A discriminator network that distinguishes between real and generated images
- Convolutional layers that capture spatial features important for eyeglass geometry

4.3.2 Dataset Preparation

The GAN was trained on a curated dataset of eyeglass frames.

4.3.3 Using Generated Images in the Try-On System

The GAN-generated eyeglass images were processed to ensure transparent backgrounds and proper scaling:

```
def make_white_transparent(image_path):
    # Load the image with alpha channel if it exists
    img = cv2.imread(image_path, cv2.IMREAD_UNCHANGED)
```

```
    if img is None:
        print(f"Error: Could not load image from {image_path}")
        return None

    # If the image doesn't have an alpha channel, add one
    if img.shape[2] < 4:
        # Convert to RGBA
        tmp = cv2.cvtColor(img, cv2.COLOR_BGR2BGRA)

        # Create a binary mask of the white background
        white_mask = np.all(img > 190, axis=2)

        # Set alpha channel to 0 for white pixels
        tmp[:, :, 3] = np.where(white_mask, 0, 255)
        img = tmp

        # ... save the processed image ...

    return img
```

Figure 12 Function to make eyeglasses' background transparent

This preprocessing step ensures that only the eyeglass frames are visible when overlaid on the video feed, creating a more realistic virtual try-on experience.

5. Evaluation and Results

5.1 Performance Metrics

The system was evaluated based on several key performance metrics:

Metric	Result	Notes
Frame Rate	25-30 FPS	Dependent on network conditions
Face Detection Accuracy	>95%	Front-facing faces under normal lighting
Eyeglass Positioning Accuracy	High	Improved with facial landmarks
System Latency	100-200ms	End-to-end from capture to display
UI Responsiveness	Good	Smooth scrolling and selection

5.2 Technical Challenges and Solutions

Several challenges were encountered during development:

5.2.1 Stable Video Streaming

Challenge: Initial implementations suffered from frequent disconnections and frame dropping.

Solution: Implemented a robust buffer management system that handles partial frames and reconnects automatically. Added error handling to gracefully recover from network interruptions.

5.2.2 Accurate Eyeglass Positioning

Challenge: Simple face detection alone resulted in unstable eyeglass positioning.

Solution: Integrated facial landmark detection to accurately identify eye positions, enabling precise eyeglass placement. Added temporal smoothing to reduce jitter.

```
# Apply simple smoothing
smooth_factor = 0.7  # Adjust between 0-1 (higher = more smoothing)

# Apply smoothing
overlay_glasses.smooth_eye_x = int(
    overlay_glasses.smooth_eye_x * smooth_factor + eye_center_x * (1 - smooth_factor))
overlay_glasses.smooth_eye_y = int(
    overlay_glasses.smooth_eye_y * smooth_factor + eye_center_y * (1 - smooth_factor))
overlay_glasses.smooth_angle = overlay_glasses.smooth_angle * smooth_factor + angle *
(1 - smooth_factor)
overlay_glasses.smooth_width = overlay_glasses.smooth_width * smooth_factor +
eye_width * (
        1 - smooth_factor)
```

Figure 13 Smoothing Code Snippet

5.2.3 Image Transparency Handling

Challenge: Eyeglass images lacked transparency channels.

Solution: Developed an algorithm to automatically detect and convert white backgrounds to transparent regions, ensuring realistic overlays.

5.2.4 User Interface Performance

Challenge: Early versions of the UI were sluggish when displaying many eyeglass options.

Solution: Implemented pagination and lazy loading of images to improve responsiveness, along with optimized image caching.

5.3 Qualitative Evaluation

User testing provided valuable qualitative feedback:

- **Realism**: Users reported that the overlay appeared convincingly attached to their face, especially with the rotation and scaling adjustments.
- **Usability**: The interface was intuitive, allowing users to quickly browse and select different eyeglass styles.
- **Performance**: The system felt responsive with minimal lag between movement and display updates.
- **GAN Images**: Users were satisfied with using GAN-generated eyeglass frames, validating the quality of the synthetic images.

8. Conclusion

The virtual eyeglass try-on system successfully demonstrates the integration of embedded systems, computer vision, and generative AI in a practical application. By combining the ESP32-S3's video streaming capabilities with advanced face detection and landmark tracking, the system provides a realistic virtual try-on experience. The incorporation of GAN-generated eyeglass frames from my master's thesis research adds significant value, addressing the limitation of diverse eyeglass frame datasets.

This project showcases the potential of edge computing devices like the ESP32-S3 for real-time augmented reality applications. The combination of affordable hardware and open-source software libraries has enabled the creation of a system that would have required specialized equipment just a few years ago.

Combining GAN-based image generation with virtual try-on systems serves as an example of how academic research can inform and support practical applications. The combination of embedded systems, computer networking, computer vision, and machine learning demonstrates a multidisciplinary approach that contributes to addressing current technological challenges.

References

1. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Nets. Advances in Neural Information Processing Systems, 27.
2. Kazemi, V., & Sullivan, J. (2014). One millisecond face alignment with an ensemble of regression trees. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1867-1874).
3. Kolkur, S., Kalbande, D., Shimpi, P., Bapat, C., & Jatakia, J. (2017). Human skin detection using RGB, HSV and YCbCr color models. Proceedings of the International Conference on Communication and Signal Processing 2016 (ICCASP 2016).
4. Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434.
5. Viola, P., & Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2001) (Vol. 1, pp. I-I). IEEE.
6. Espressif Systems. (2021). ESP32-S3 Technical Reference Manual. https://www.espressif.com/sites/default/files/documentation/esp32-s3_technical_reference_manual_en.pdf
7. Zhang, K., Zhang, Z., Li, Z., & Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. IEEE Signal Processing Letters, 23(10), 1499-1503.
8. G. Shubitidze, "Virtual Eyeglass Try-On System Using ESP32-S3 and Computer Vision," GitHub repository, 2025. [Online]. Available: https://github.com/bokuwagiga/virtual_eyeglasses_tryon