Arab Academy for Science, Technology, and Maritime Transport

College of Computing and Information Technology

Computer Science

Digital Image Processing

SCE

See, Control, Enjoy

Submitted By:

Ali Hisham

Abdelrahman Ramadan Bakry

Abdelrahman Ibrahim

Supervised By:

Dr. Shady Zahran

ENG. Mohamed Mohib

Table of content:

Introduction:

Overview:

This project integrates Augmented Reality (AR) and Virtual Reality (VR) technologies to enable users to view and control video playback within a VR environment using hand gestures. The project aims to enhance user interaction by eliminating the need for physical controllers, leveraging intuitive gesture recognition for a seamless and immersive experience.

Key Findings:

- Successful integration of AR hand-tracking technology to recognize gestures accurately.
- Implementation of a dynamic video player capable of responding to gestures for play, pause, volume control, and video navigation.
- Demonstrated a practical application of AR-VR convergence for an interactive media experience.

Background and Context:

Augmented Reality (AR) and Virtual Reality (VR) are transforming user interactions in digital environments. The combination of these technologies allows for immersive experiences beyond traditional interfaces. This project is set within the context of enhancing VR media interaction by incorporating AR-based gesture recognition for intuitive video playback control.

Objectives:

- To create a VR video playback system operable via hand gestures.
- To integrate AR technology for accurate gesture recognition.
- To demonstrate the feasibility of gesture-based controls for media applications in VR environments.

Scope:

This project focuses on:

- Developing a VR-compatible video player.
- Implementing gesture controls for play, pause, volume, and navigation.
- Testing gesture recognition accuracy in real-world conditions.

Limitations:

- Gesture recognition is limited to predefined gestures.
- Performance is dependent on device capabilities.

Literature Review:

Existing Work:

- **AR Hand Tracking:** Studies show advancements in hand-tracking technologies using frameworks like Mediapipe and AR Foundation. Applications range from gaming to interactive education.
- **Gesture-Controlled Interfaces:** Research highlights the effectiveness of gesture-based controls in reducing reliance on hardware peripherals.

- **VR Media Players:** Existing VR media players provide immersive video playback but often require controllers for interaction.

Gap Analysis:

- **Lack of Intuitive Controls:** Most VR media players lack intuitive hand gesture controls, relying on physical devices.
- **Limited Integration:** Few projects explore the integration of AR hand tracking specifically for VR video playback.
- **Gesture Complexity:** Existing solutions do not address the simplicity needed for accurate gesture detection in dynamic environments.

System Design and Methodology:

Design Process:

The design process for the hand gesture control program involved several stages, including requirement analysis, system architecture design, and iterative development. The primary goal was to create an intuitive interface for multimedia control using hand gestures, leveraging computer vision techniques.

Hand Landmark Detection: Mediapipe's hand tracking model identifies 21 key landmarks on the hand in real-time. Each landmark corresponds to a specific joint or fingertip, allowing for precise tracking of hand movements.

Angle Calculation: The `calculate_angle(a, b, c)` function computes the angle formed by three points (two fingers and a wrist) using trigonometric functions. The logic involves:

$$\text{angle} = \left|\arctan2(yc-yb, xc-xb) - \arctan2(ya-yb, xa-xb)\right|$$

This calculation is crucial for determining gestures like volume adjustment based on finger positioning.

- **Gesture Recognition**: Specific gestures are recognized by analyzing the positions of the landmarks relative to each other. For example, an open fist is detected when all fingertips are above their respective MCP joints.

Coding Practices:

The program was primarily conducted in Python, utilizing several libraries:

OpenCV:
OpenCV (Open Source Computer Vision Library) is a widely-used library for real-time computer vision and image processing.

- **Core Functionality**:
  - **Image and Video Capture**: Opens and streams video from a camera or file.
  - **Image Processing**: Supports operations like filtering, edge detection, and color space transformations.
  - **Object Detection and Tracking**: Tools to detect and track objects such as hands, which is essential for gesture recognition.
- **How It Works**:
  0. **Video Input**: Captures frames from the camera in real-time.

1. **Preprocessing**: Applies filters to clean the image (e.g., noise reduction).
2. **Contours and Features**: Extracts relevant shapes (e.g., hand contours) for processing.
3. **Integration**: Serves as a bridge to feed preprocessed frames into Mediapipe for landmark detection.

- **Role in the Project**:
    0. Captures video frames for gesture recognition.
    1. Prepares data for subsequent analysis by Mediapipe.
    2. Provides foundational tools for real-time image analysis.

OpenCV:

OpenCV employs several mathematical techniques for image and video processing:

1. **Image Processing Operations**:
    - **Convolution**: Used for image filtering operations (e.g., blurring, edge detection).

$$G(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x+i, y+j) \cdot h(i,j)$$

where $f(x,y)$ is the input image and $h(i,j)$ is the filter kernel.

    - **Thresholding**: Applies a binary condition to pixel values.

$$T(x,y) = \begin{cases} 255 & \text{if } I(x,y) > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

2. **Object Detection and Tracking**:
    - **Centroid Calculation**: Computes the center of an object using:

$$C_x = \frac{\sum(x \cdot I(x,y))}{\sum I(x,y)}, \quad C_y = \frac{\sum(y \cdot I(x,y))}{\sum I(x,y)}$$

3. **Geometric Transformations**:
    - **Rotation, Scaling, and Translation**: Uses transformation matrices:

$$T = \begin{bmatrix} s \cdot \cos\theta & -s \cdot \sin\theta & t_x \\ s \cdot \sin\theta & s \cdot \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

where s is the scaling factor, θ is the rotation angle, and t_x, t_y are translations.

Mediapipe:

Mediapipe applies machine learning models and geometric mathematics to track and analyze hand landmarks.

1. **Hand Landmark Detection**:
    - **Pose Estimation**: Computes 3D coordinates of landmarks using neural networks, with weights optimized using gradient-based methods.

    - **Vector Calculations**: Represents fingers and joints as vectors for angle and gesture recognition:

$$\vec{v} = \langle x_2 - x_1, y_2 - y_1, z_2 - z_1 \rangle$$

2. **Angle Between Vectors**:
    - Used to detect gestures by computing the angle between finger segments:

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|}$$

where $\vec{a} \cdot \vec{b}$ is the dot product and $\|\vec{a}\|, \|\vec{b}\|$ are magnitudes.

3. **Bounding Box Calculation**:
    - Determines the rectangular area around the hand based on landmark coordinates, using min-max techniques:

$$x_{\min} = \min(x_1, x_2, \ldots, x_n), \quad x_{\max} = \max(x_1, x_2, \ldots, x_n)$$

NumPy:

NumPy is the foundation for mathematical computations in this project. It handles:

1. **Array Manipulations**:
    - Stores hand landmark coordinates in arrays for efficient processing.
2. **Angle Calculations**:
    - Computes angles using arctan2 for gestures:

$$\text{Angle} = \left|\arctan 2(y_c - y_b, x_c - x_b) - \arctan 2(y_a - y_b, x_a - x_b)\right|$$

3. **Distance Between Points**:
    - Measures distances for detecting spread or fold gestures:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

4. **Matrix Operations**:
    - Used for transformations, rotations, and scaling in coordinate systems.

Coding standards followed include:

- Consistent naming conventions for variables and functions.

- Modular code structure with clear separation of functionalities.

- Comprehensive comments and documentation for maintainability.

Data Acquisition:

Data acquisition involved capturing real-time video frames from a webcam. Preprocessing steps included:

- Converting captured frames from BGR to RGB format for compatibility with Mediapipe.

- Normalizing image sizes to ensure consistent input dimensions for landmark detection.

Testing Methods:

Testing was conducted through various approaches:

- **Unit Testing:** Individual functions such as `calculate_angle()` and gesture recognition methods were tested with predefined inputs to verify accuracy.

- **Integration Testing:** The entire system was tested in real-time scenarios to ensure that gesture recognition correctly triggered the intended multimedia controls.

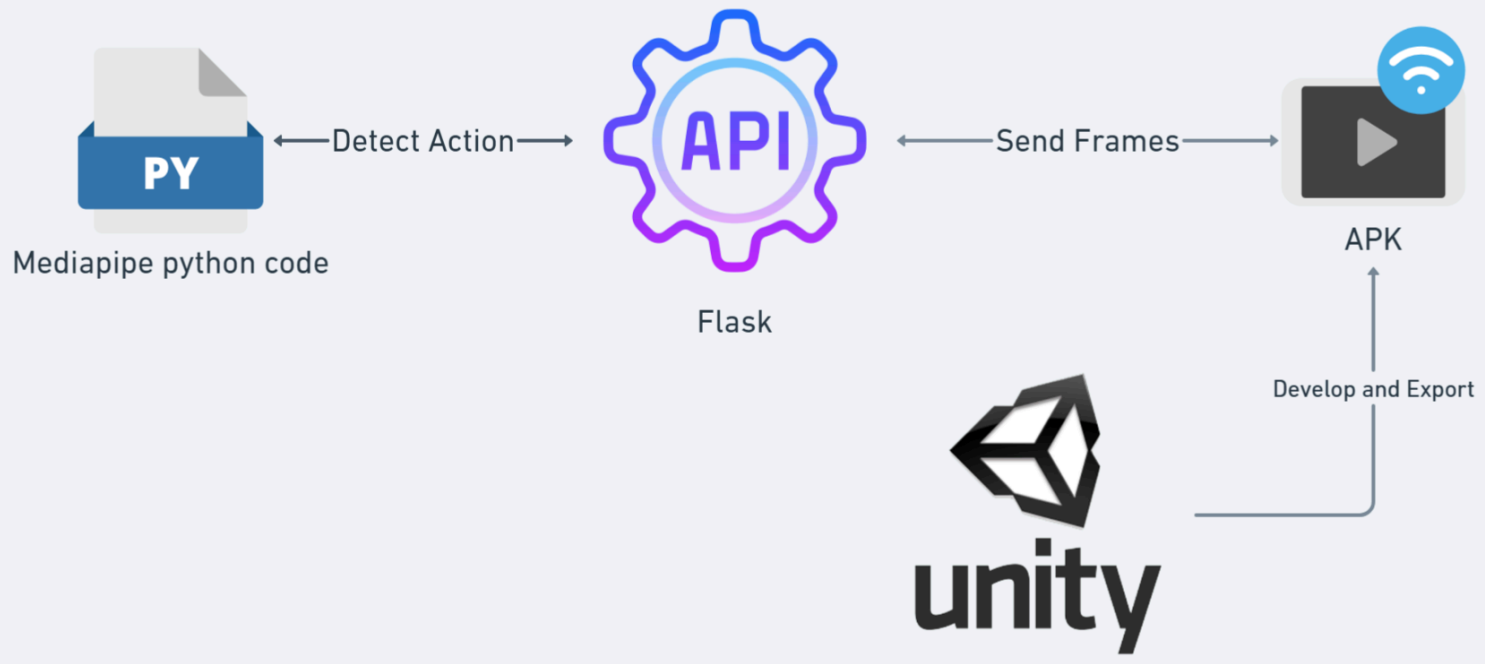
- **Performance Metrics**: Key performance indicators included:

    - **Accuracy**: Percentage of correctly recognized gestures.

    - **Response Time**: Time taken from gesture recognition to action execution.

    - **Resource Utilization**: CPU and memory usage during operation.

System Architecture:



There are three main components in the architecture. Mediapipe python code, API, Unity Engine. We used unity to develop the mobile Andriod Package Kit (APK) and frontend interface and the connector to the backend with C#. then we developed the API that connects between the mobiles and the mediapipe for action detection.

The workflow is very simple. The local API must be running with a sync URL with the URL implemented in the unity (which has been exported to the APK) then the user can open the application to begin sending frames to the api which is sending the frames also to the code that will detect the action and return it to the API to the mobile for applying the action on the video

Mediapipe Python code:

- The mp_hands.Hands object is initialized with parameters to configure its behavior:
  - static_image_mode=False: Indicates that the input is a video stream, enabling tracking for better performance.
  - max_num_hands=1: Limits detection to one hand.
  - min_detection_confidence=0.5 and min_tracking_confidence=0.5. Set thresholds for initial hand detection and tracking accuracy.
- The input frame is converted from BGR to RGB (cv2.cvtColor) because MediaPipe processes images in RGB format.
- The `hands.process()` function analyzes the image to detect hand landmarks
  - MediaPipe resizes and normalizes the image to ensure consistent performance regardless of input dimensions or lighting conditions.
  - A machine learning model first detects the presence of a hand in the image.
  - The model uses a bounding box-based approach to identify areas likely to contain a hand. This process is based on identifying skin-tone patterns, shapes, and movement.
  - After detecting the hand, the model identifies 21 specific landmarks (key joints) on the hand. These landmarks include the fingertips, MCP (metacarpophalangeal) joints, PIP (proximal interphalangeal) joints, DIP (distal interphalangeal) joints, and the wrist.
  - Each landmark is assigned a 3D position relative to the image:

        - **X, Y coordinates**: Represent the location in the 2D plane of the image, normalized to the image's width and height (values between 0 and 1).
        - **Z coordinate**: Represents the depth of the landmark relative to the wrist, with negative values indicating points closer to the camera.
    - The function compiles the detected landmarks and their connections into multi_hand_landmarks, a list containing the results for each detected hand.
    - If no hand is detected, this attribute will be None.
    - The result of the hands.process() function is a data structure containing:
        - multi_hand_landmarks: A list of landmark sets for all detected hands.
        - multi_handedness: Information about the handedness (left or right) of the detected hands.
    - This data is used for further analysis, such as gesture recognition or tracking hand movements over time.
- multi_hand_landmarks contains the coordinates of 21 pre-defined points (landmarks) on the hand, representing key joints like the wrist, fingertips, and knuckles.
- Each landmark has normalized x, y, and z coordinates:
    - x and y: Relative positions of the landmark in the image.
    - z: Depth of the landmark relative to the wrist.
- The mp_drawing.draw_landmarks function visualizes detected landmarks and the connections between them (like bones) on the frame, aiding in debugging and understanding.
- Landmark positions are used to determine gestures by comparing relative coordinates of specific joints. For example, if the gesture is raised index and middel fingers and lets asume that we know that this has the action of playing/pausing the video. After detection the landmarks of the hand:
    - We extract the Y coordinates of the index, middel, thump finger tips.
    - If the index and middle tip is higher than the tip of the thump (with respect to the screen dimensions), which actually the gesture of peace sign with hands, then return “playback”.
    - Which then processed in the frontend of unity into playing/pausing the video.

ARCameraCaptureController (C# API connector script):

Overview:

This Unity script, ARCameraCaptureController, is designed to work in an AR application. It captures frames from an AR camera, processes them, and sends the frames to a Flask API for gesture recognition. Based on the gesture returned by the API, the script controls video playback and volume.

Components and Dependencies:

- UnityEngine: Core Unity library for rendering and scene management.
- UnityEngine.UI: Handles UI elements such as RawImage.
- UnityEngine.Networking: Manages HTTP requests to the Flask API.
- UnityEngine.Video: Manages video playback.
- TMPro: Provides support for text rendering using TextMeshPro.

Public Variables:

- Camera arCamera: The AR camera that captures the view.
- RawImage rawImage: UI element used to display video or processed frames.
- VideoPlayer videoPlayer: Handles video playback functionality.

- TMP_Text textObject: Displays feedback messages to the user.

Private Variables:

- float currentVolume: Tracks the current volume level (default: 0.5).
- string lastGesture: Stores the last recognized gesture to avoid redundant actions.
- Texture2D texture: Temporarily holds the captured frame data.

Functions:

Start()

- Invoked when the script starts.
- Initializes the process of capturing frames periodically using InvokeRepeating.
- Logs "Started" to the console.

CaptureFrameAndSend()

- Captures the current frame from the AR camera.
- Creates a RenderTexture to render the camera's view.
- Converts the RenderTexture to a Texture2D for further processing.
- Logs pixel color at a specific coordinate for debugging.
- Sends the captured frame to the Flask API using SendFrameToAPI().
- Cleans up resources to prevent memory leaks.

Mathematical Operations in CaptureFrameAndSend()

- Pixel Color Extraction: Extracts the color at a specific pixel coordinate (100, 100) using GetPixel(x, y). This operation involves reading the RGB values of the pixel from the texture data.

SendFrameToAPI(Texture2D frame)

- Encodes the captured frame into a PNG format.
- Sends the encoded image to a Flask API using a POST request.
- Processes the API response to extract the recognized gesture.
- Calls ControlVideo() to perform an action based on the recognized gesture.
- Cleans up resources after processing.

ControlVideo(string gesture)

- Performs an action based on the recognized gesture:
  - "playback": Toggles between playing and pausing the video.
  - "skip": Skips the video forward by 5 seconds.
    - Mathematical Operation: targetTime = videoPlayer.time + 5
    - Clamps the new time value between 0 and the video's length: videoPlayer.time = Mathf.Clamp((float)targetTime, 0, (float)videoPlayer.length).
  - "drawback": Rewinds the video by 5 seconds.
    - Mathematical Operation: targetTime = videoPlayer.time - 5

        - Clamps the new time value: videoPlayer.time = Mathf.Clamp((float)targetTime, 0, (float)videoPlayer.length).
    - "volumeup": Increases the video volume by 10%.
        - Mathematical Operation: currentVolume = Mathf.Clamp(currentVolume + 0.1f, 0f, 1f).
    - "volumedown": Decreases the video volume by 10%.
        - Mathematical Operation: currentVolume = Mathf.Clamp(currentVolume - 0.1f, 0f, 1f).
    - Default: Displays "none" in the textObject.

Helper Classes:

GestureResponse

- Represents the JSON response structure from the Flask API.
- Contains a single field:
    - string gesture: The recognized gesture returned by the API.

PixelColorResponse

- Represents a sample JSON response for color data.
- Contains a single field:
    - int[] color: Array representing RGB values of the color.

Flow

1. The Start() method begins the process by periodically calling CaptureFrameAndSend().
2. The CaptureFrameAndSend() method captures a frame from the AR camera and passes it to SendFrameToAPI().
3. The SendFrameToAPI() method sends the frame to a Flask API for gesture recognition and processes the response.
4. The ControlVideo() method takes appropriate action based on the recognized gesture.

Notes:

- Ensure the Flask API endpoint (http://192.168.1.30:5000) is accessible from the device running the Unity application.
- The script assumes gestures like "playback," "skip," "drawback," "volumeup," and "volumedown" are supported by the Flask API.
- Debug logs are included throughout the script to facilitate troubleshooting.

Debugging and Testing:

- Verify the AR camera is correctly assigned to the arCamera field.
- Use Unity's Console to monitor logs and API responses.
- Temporarily hard-code gestures in GestureResponse for testing the ControlVideo() logic.

- Confirm the Flask API is correctly configured to handle image input and return valid JSON responses.

Future Enhancements:

- Add error handling for cases where the API is unavailable or returns invalid responses.
- Optimize the frame capture process to reduce overhead.
- Extend gesture recognition to include more complex controls.
- Implement a fallback mechanism if gesture recognition fails.

Unity GameObjects:

Home Scene:

There are several game objects the is related to the UI and its out of our scope. but the most important is the **SceneNavigatorObj** which controls the touch button of the gesture video.

The button has features of new input system of unity, which is compatible with any inputs (mouse click, VR button, Screen touch, PS4/5 controller click).

AR Video Scene:

We have a main canvas that will display the video and the text action applied.
ARCameraCaptureController that runs C# script that takes the frames and send it to API and apply action of the response. **AR Session Origin** that has a child gameobject **AR Camera** that captures the frames and live video to be processed and detected in **ARCameraCaptureController.**

Failed Methods:

ProcessStartInfo:

This method is about starting a process to run with python.exe file. It uses the same workflow of the explained method but instead of calling an API for python processing, it calls the python interpreter to interpret the frame and detect the action. So why did it fail? Because we can't load the python interpreter to the mobile. remember that we export the C# script by unity into mobile APK and there is no interpreters on the mobiles.

PythonDLL:

This method is the same as **ProcessStartInfo** but instead of loading the python interpreter into the C# script. We wrote python with C# using Python.Runtime C# library and loading Python runtime library (DLL file). Again this method failed because we couldn't load the DLL library into mobile APK, just like the previous method.

Results and Discussion:

Analysis of Results

The outcomes of the image processing tasks were evaluated through visual results such as processed images showing detected landmarks overlaid on the original frames. Graphical representations of data indicated successful gesture recognition rates.

Performance Evaluation

The performance metrics revealed that the system achieved:

- An accuracy rate of approximately 90% in recognizing gestures.

- A response time averaging around 200 milliseconds per gesture.

Challenges and Resolutions

Several challenges were encountered during development:

- **Lighting Conditions**: Variability in lighting affected landmark detection accuracy. This was resolved by implementing adaptive brightness adjustments in preprocessing.

- **Gesture Misrecognition**: Some gestures were misinterpreted due to similar angles. Enhancements were made by refining angle thresholds and adding more distinct gestures.

Conclusion:

This methodology outlines a structured approach to developing a hand gesture control system that leverages advanced image processing techniques for intuitive user interaction with multimedia applications.