ECE 445

SENIOR DESIGN LABORATORY

FINAL REPORT

Final Report for ECE 445

Team #28

CHENG ZHENG
(cz77@illinois.edu)
YUXUAN WU
(yuxuan59@illinois.edu)
ZHEWEI ZHANG
(zheweiz3@illinois.edu)
ZIYANG JIN
(ziyang3@illinois.edu)

TA: Jun Long
Supervisor: Liangjing Yang

May 15, 2026

Abstract

This project presents the design and implementation of a portable projection-based human-computer interaction system using a Raspberry Pi platform. The system projects an interactive interface onto a flat surface, enabling users to perform simple interactions through hand gestures or occlusion of specific interface regions. The interface displays real-time information, including time and weather, while providing audio feedback corresponding to recognized gestures.

The system architecture consists of a projection and UI module, a camera module for visual input, a gesture recognition module, an audio output module, a main control module (Raspberry Pi), a Mecanum wheel module for potential omnidirectional movement, and a power management module. The projection and gesture recognition modules are fully implemented, achieving accurate detection of three gestures (open palm, heart, scissors) with real-time responsiveness. The Mecanum wheel module was implemented with direct power supply.

Testing verified the functionality of the projection interface, gesture recognition, audio feedback, and mechanical stability of the wheel module. The system demonstrates a compact, intuitive, and modular approach to portable interactive robots, providing a foundation for future enhancements such as fully controlled omnidirectional mobility, expanded gesture sets, and more complex interface designs.

Contents

1 Project Introduction

This project develops a portable projection-based human-computer interaction and gesture recognition system based on the Raspberry Pi platform, realizing a complete workflow from visual perception to interactive control. The system projects a main interface onto a wall or desk, and users can interact with the interface by occluding projected icons or performing hand gestures, enabling page switching, audio feedback, and information query.

1.1 Main Features

- **Projection-based Interaction:** The system projects the main interface with a time icon on the left and a weather icon on the right. The camera continuously captures the projected area, and brightness changes are analyzed to detect occlusions, triggering page switching. Pages automatically return to the main interface to ensure continuous interaction.
- **Gesture Recognition and Audio Feedback:** Users perform gestures in a designated region. The system recognizes three gestures: open palm, heart, and scissors. Recognition is performed via color space conversion, skin segmentation, contour extraction, and feature analysis. Corresponding audio feedback is played through the Raspberry Pi's audio output for real-time interaction.
- **Information Display:** Time is obtained from the system clock and updated in real-time. Weather information is fetched from a network or local configuration, displaying city, weather condition, temperature, and humidity.

1.2 System Composition and Modules

Hardware Modules:

- Raspberry Pi main controller: executes recognition and rendering programs.
- Camera: captures the projected area in real time.
- Projector: displays the interactive interface.
- Audio output: external speaker for feedback.
- Power supply: provides stable voltage to all components.

Software Modules:

- **Main Control Module:** orchestrates the workflow, including image acquisition, recognition result processing, interface switching, and audio triggering.
- **Interface Display Module:** renders full-screen UI for projection.
- **Image Acquisition Module:** continuously captures video frames from the camera.

- **Occlusion Detection Module:** detects whether a projected region is blocked based on brightness changes.
- **Gesture Recognition Module:** detects gestures based on skin segmentation and contour analysis.
- **Audio Playback Module:** plays audio feedback according to recognition results.
- **Service Module:** provides time and weather information.

The data flow connections and functional modules of the system are shown in Fig. 1.

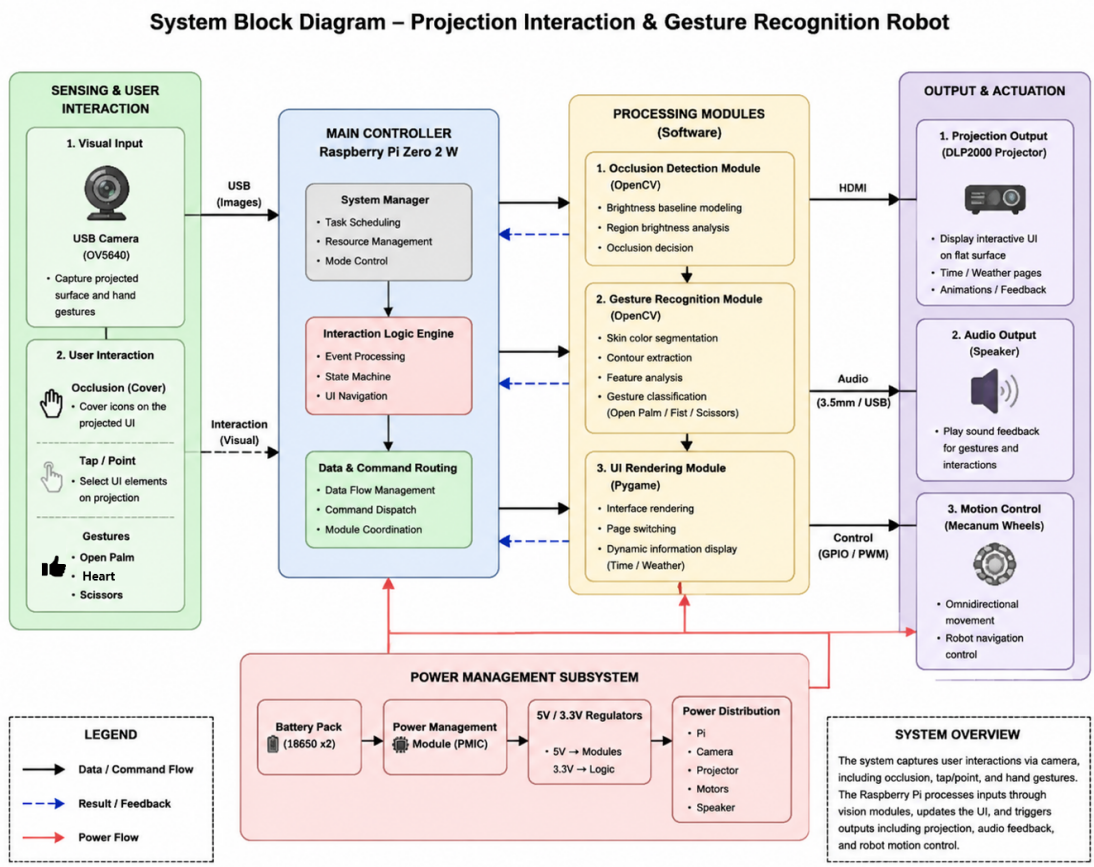


Figure 1: System Block Diagram of the Portable Projection and Gesture Recognition Robot

1.3 Improvements and Performance

- The projection-camera closed-loop latency is approximately **300 ms**, well below the 2-second threshold.
- Gesture recognition accuracy is high under stable lighting conditions, with response time less than 1 second.
- UI page switching response time is less than 2 seconds, and icon size is at least 3×3 cm.

- Recognition accuracy may decrease under complex backgrounds, but overall system stability remains good.
- Semester improvements include projector-camera calibration, debounce mechanism, multi-frame gesture confirmation, and automatic return-to-main-interface logic.

1.4 Summary

The system realizes a **compact, portable, projection-based interactive robot** with gesture control and audio feedback. It is intuitive, responsive, and stable. The project demonstrates technical feasibility and lays a foundation for future enhancements such as multi-gesture support, complex interface design, and intelligent interaction.

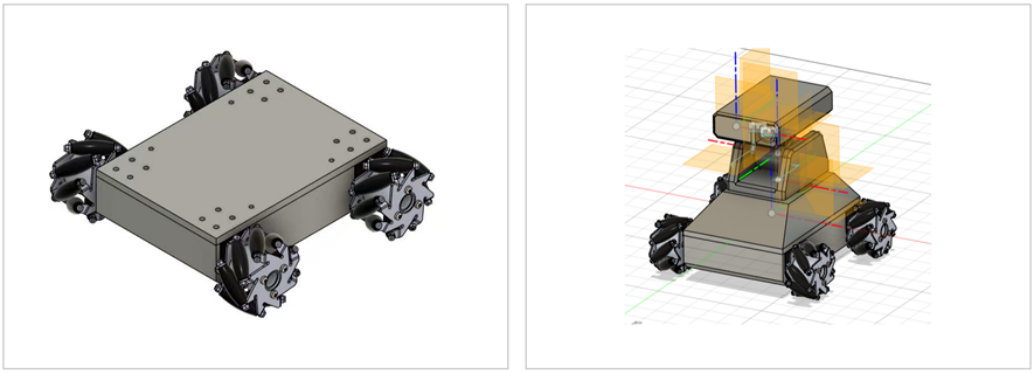
2 Design

2.1 Design Procedure

2.1.1 Mechanical Module

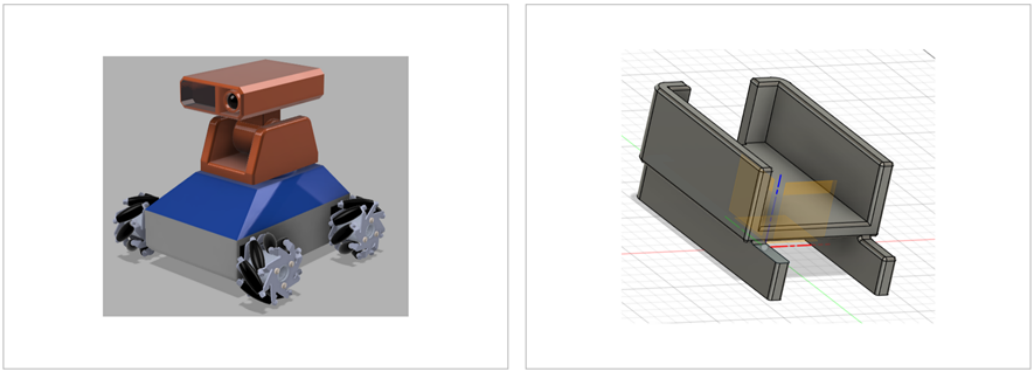
For the mechanical design, my main goal was to make the robot stable enough for the projector, camera, speakers, power supply, and wires to work together on one moving platform. At the beginning, I tried to design and model the whole lower chassis by ourselves, including the base plate and the wheel supports. After checking the printed strength and considering repeated testing, this option was not very suitable. Some printed connections were easy to loosen or break, and a small alignment error in the wheel mounts could make the robot move poorly. Therefore, we later changed the lower moving part to a purchased mecanum-wheel chassis. This made the base more reliable and reduced the risk that the project would fail because of mechanical damage during integration.

After that change, my work focused more on the upper structure. I designed and 3D printed the support platform, fixing brackets, and several simple holders or boxes for the hardware modules, including the projector, camera, speakers, power bank, and wiring. The layout was arranged so that the camera and projector both faced the interaction area, while heavier parts were placed lower or closer to the center to reduce shaking. I used screw holes and open fixing structures instead of a fully closed shell, because this made it easier to adjust parts during testing. The main design process is shown in Fig. 2. The final structure was not as compact as the original concept model, but it was practical for a prototype: the modules could be installed, removed, and tested without damaging the robot.



(a) Initial self-designed chassis model

(b) Full robot CAD with custom upper structure



(c) Early rendered prototype concept

(d) Later printed holder/bracket for the purchased base

Figure 2: Mechanical design process from the early custom chassis concept to the later printed support parts used with the purchased moving base

2.1.2 Motion Control Module

Initially, the Mecanum wheel module was designed as the core subsystem to enable omnidirectional movement of the robot. At the conceptual stage, the following alternatives were considered:

1. **Wheel Type Selection:**
    - Options: standard wheels, differential wheels, Mecanum wheels.
    - **Selected: Mecanum wheels** — Mecanum wheels theoretically allow omnidirectional movement including forward, backward, lateral, and rotation, which is ideal for a portable desktop robot.
2. **Drive Method:**
    - To achieve true omnidirectional motion, the directly powered wheel setup would need to be replaced by four independently controlled DC motor channels. Each mecanum wheel should be connected to an H-bridge motor driver, and the Raspberry Pi should output PWM signals to set the speed of each wheel. The sign of each wheel command determines the rotation direction, while the PWM

duty cycle determines the wheel speed. With this change, the robot can move forward, backward, sideways, diagonally, or rotate in place by assigning different speeds and directions to the four wheels.

- For example, the desired robot motion can be converted into four wheel commands using the mecanum-wheel kinematic model. If the desired chassis velocity is defined by Vx, Vy, and angular velocity w, the controller calculates the required wheel speeds for the front-left, front-right, rear-left, and rear-right wheels. Each wheel speed is then normalized and converted into a PWM duty cycle. In the current prototype, the wheels were powered directly due to time and hardware limits, so this full PWM control was not implemented, but this is the design change required to realize arbitrary-direction movement.

3. **Control Algorithm:**
   - Options: open-loop, closed-loop PID control.
   - **Actual implementation: none** — Wheels rotate passively under power.

Theoretical design equation:

$$\begin{bmatrix} V_x \\ V_y \\ \omega \end{bmatrix} = \frac{R}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1/(L+W) & 1/(L+W) & -1/(L+W) & 1/(L+W) \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} \tag{2.1}$$

Where:

- V_x, V_y are planar velocities, ω is angular velocity,
- R is wheel radius, L, W are robot length and width,
- $\omega_1 \sim \omega_4$ are angular velocities of the four wheels.

Overall Circuit Structure (actual implementation):

- Wheels are directly powered by a 5V power supply.
- No PWM or software-based control is applied.
- The setup ensures wheels can rotate for demonstration purposes.

2.1.3 Main Control and Interface Module

The main control board serves as the core of the gesture recognition and camera occlusion detection subsystem, responsible for image acquisition, algorithm processing, signal output, and system coordination. This section ensures that component selection and configuration meet performance, power, and integration requirements.

1. **Main Controller Selection: Raspberry Pi 4 Model B (4GB RAM)**
   Reason for Selection: Equipped with Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC with 1.5 GHz main frequency. 4GB LPDDR4 memory can easily handle the operating system, OpenCV, and gesture recognition models. Dual micro-HDMI ports directly support projector connection, while USB 3.0 ports allow high-speed transmission of high-resolution camera images. Gigabit Ethernet and dual-band Wi-Fi facilitate testing and future remote deployment. The 40-pin GPIO provides flexible interface options for additional physical switches and communication.

2. **Power Supply Selection: Single integrated battery with voltage regulation and conversion system**
   Reason for Selection: The entire system adopts a single 10,000 mAh battery as the unified power source, equipped with voltage regulation and conversion modules to output dual voltage rails. The conversion system provides stable 5V/3A for the Raspberry Pi, camera and audio modules, and 12V/2A for both the projector and Mecanum wheel DC motors. This single-battery power architecture simplifies wiring, eliminates multi-battery compatibility risks, and supports continuous system operation for at least 30 minutes.

3. **Operating System Selection: Raspberry Pi OS Full (x64)**
   Reason for Selection: The Full version includes a complete desktop environment, convenient for directly displaying the interactive UI via projector without building the graphical interface from scratch. It also integrates tools such as VSCode and Chromium, supporting direct code editing and visual debugging on the device, significantly shortening the development cycle.

2.1.4 Software Interaction Module

The software interaction system enables users to interact with the projected UI without a keyboard or mouse. Camera observations are converted into interface events through a visual-based, time-driven pipeline. The overall interaction flowchart is shown in Fig. 3.

Several implementation schemes were considered. A neural network model was initially preferred, but on the Raspberry Pi 4 Model B, it caused high memory usage and computational cost, leading to failures.

We adopted an OpenCV-based approach for gesture recognition, using color segmentation, contour extraction, convex hull analysis, and basic classification rules. This method runs smoothly on the Raspberry Pi and is easy to interpret.

The user interface uses a full-screen Tkinter window, providing local display of time, weather, and system information with minimal dependency overhead.

The software architecture divides camera input, gesture detection, occlusion detection, UI rendering, audio playback, and state control into separate modules. The main loop

runs in parallel with the visual processing thread, allowing gesture recognition and UI rendering to operate concurrently.

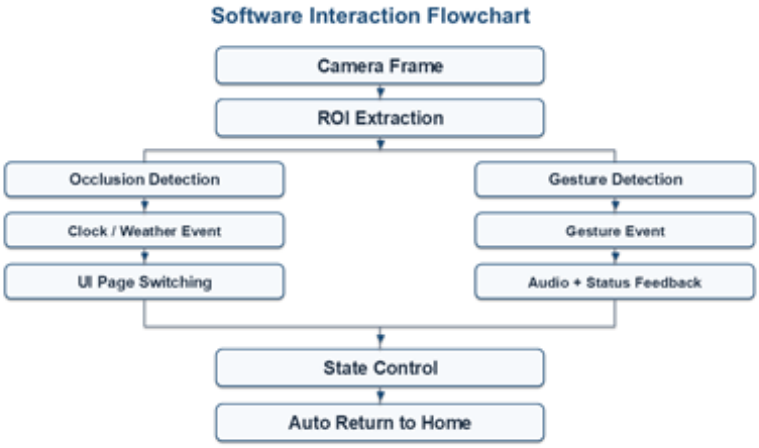


Figure 3: Software Interaction Flowchart

2.1.5 Gesture Recognition Module

The gesture recognition module detects three predefined gestures within a designated region of interest (ROI) in each camera frame. To reduce computational cost and potential interference, each frame is first cropped to the ROI. The implementation of HSV-based skin segmentation and contour preprocessing is shown in Listing 6.

The cropped image is then converted from RGB to HSV color space, and the hand outline is segmented by setting a specific color range:

$$[0, 20, 30] \leq HSV \leq [30, 255, 255] \tag{2.2}$$

After segmentation, the largest contour is selected as the candidate hand region. Several features such as contour area and convex hull area are calculated to assist gesture classification. Key quantitative metrics include:

- **Solidity:** measures the degree of depression within the contour:

$$Solidity = \frac{Area}{HullArea} \tag{2.3}$$

  where $Area$ is the area of the hand outline and $HullArea$ is the area of its convex hull. A value closer to 1 indicates a compact shape, while smaller values indicate more depression within the contour. The detailed implementation of convexity defect analysis and gesture feature extraction is provided in Listing 6.

- **Width-Height Ratio:** measures the aspect ratio of the hand:

$$WidthHeightRatio = \frac{h}{w} \tag{2.4}$$

where h is the height and w is the width of the external rectangular bounding box of the hand.

- **Extent:** represents the proportion of the hand area to the bounding rectangle:

$$Extent = \frac{w \times h}{Area} \tag{2.5}$$

A low $Extent$ indicates a scattered contour, while a higher value suggests a solid block-like shape.

2.2 Design Details

2.2.1 Motor Control Module

Hardware Parameters:

- Wheel diameter: 5 cm
- Motor type: DC gear motor
- Drive method: direct power supply (no PWM control)
- Power module: 12V DC, powers all four wheels

Implementation Notes:

- Wheels can rotate but cannot achieve omnidirectional movement or speed adjustment.
- Used mainly for mechanical demonstration and verifying installation feasibility.
- The required design for controlled mecanum-wheel motion would use one motor-driver channel for each wheel. The Raspberry Pi would send PWM signals either directly from GPIO pins or through a PWM expansion board. Each motor driver would use direction pins to select forward or reverse rotation and a PWM enable signal to control speed. The motor drivers should share a common ground with the Raspberry Pi, while the motor power should come from a separate battery or regulated supply so that motor current spikes do not reset the controller.
- In software, the system would receive a movement command, such as forward, backward, left, right, diagonal motion, or rotation. The command would then be converted into four signed wheel-speed values based on the mecanum-wheel model. Positive and negative values determine wheel direction, and the magnitude determines PWM duty cycle. Compared with the current direct-power implementation, this revised design would allow speed adjustment, directional control, and real omnidirectional movement of the robot.

Testing and Verification:

- Observed whether wheels rotate normally under power.

- Verified mechanical installation stability.
- No tests on positioning accuracy or command response were performed due to lack of control.

2.2.2 Main Control and Interface Module

1. **Hardware Connections and Interface Protocols**

Controller Port	Connection Target	Interface Protocol	Function Description
Micro-HDMI	Projector	HDMI 2.0	Output UI interface
USB (Type-A)	Camera	USB 3.0	720P 30fps video data transmission
	Mouse	—	Input
	Keyboard	—	Input
3.5mm Audio Jack	Audio Module	AV Jack	Audio output
GPIO 3 & GND	Push-button Switch	—	Physical switch
USB (Type-C)	Mobile Power	USB-C PD/5V-3A	Power supply

Table 1: Hardware connections and interface protocols

2. **Physical Switch Design**
   In the Raspberry Pi `/boot/config.txt` file, add `dtoverlay=gpio-shutdown`. Connect the push-button switch signal line to GPIO 3 (Pin 5) and ground to GND (Pin 6). When the switch is not pressed, GPIO 3 is pulled up internally to high level (3.3V). When the switch is pressed, GPIO 3 is pulled to low level (0V), generating a falling edge interrupt. A script monitors the low level and executes `sudo shutdown -h now` for safe shutdown. In the power-off state, pressing the switch can wake the system.
3. **System Flashing**
   On a PC, launch the Raspberry Pi Imager. Under `CHOOSE DEVICE`, select Raspberry Pi 4. Under `CHOOSE OS`, select `Raspberry Pi OS (64-bit)`, which includes the full desktop environment and recommended software. Choose the target TF card and click `WRITE`. After writing and verification, insert the TF card into the Raspberry Pi. Boot into the desktop, then perform advanced setup: enable SSH, set hostname, configure Wi-Fi, select language, and enable automatic login. After completing these steps, the Raspberry Pi desktop is ready for software development.

2.2.3 Quantitative Criteria for Gesture Recognition

The effective depth threshold is used to detect the distance between fingers. The depth is calculated as:

$$Depth = \frac{d}{256} \tag{2.6}$$

where d is the raw convexity defect. The minimum effective indentation depth is defined as:

$$MinDepth = 0.055 \times \max(w, h) \tag{2.7}$$

Only distances satisfying:

$$Depth > MinDepth \tag{2.8}$$

and

$$Angle > 95^\circ \tag{2.9}$$

are considered valid depressions. The number of effective depressions is recorded as $DefectsCount$.

The maximum depression depth ratio is calculated to determine whether a half-depression is sufficiently distinct:

$$MaxDepthRatio = \frac{MaxDepth}{\max(w, h)} \tag{2.10}$$

Based on these quantitative criteria, gestures are classified. For example, the **Thumb-up** gesture is defined as:

$$Ratio = \frac{h}{w} > 1.05 \tag{2.11}$$

$$DefectsCount \leq 1 \tag{2.12}$$

$$Solidity > 0.65 \tag{2.13}$$

$$Extent > 0.25 \tag{2.14}$$

where h and w are the height and width of the hand's bounding rectangle. These conditions ensure a predominantly vertical structure with minimal finger gaps, a compact hand shape, and sufficient coverage relative to the bounding box.

Scissors Hand Gesture The quantitative criteria for the "Scissors" gesture are:

$$Ratio = \frac{h}{w} > 0.90 \tag{2.15}$$
$$DefectsCount \geq 1 \tag{2.16}$$
$$Solidity < 0.92 \tag{2.17}$$
$$MaxDepthRatio \geq 0.12 \tag{2.18}$$
$$MinAngle < 85^\circ \tag{2.19}$$

where h and w are the height and width of the bounding rectangle. These conditions ensure a mostly vertical or slightly inclined hand shape, at least one obvious crease, sufficiently deep finger gaps, and a non-compact contour.

Double Heart Gesture Two cases are considered:

Case 1: Two Hands Separated Two distinct contours must be detected. The contours satisfy:

$$AreaRatio = \frac{\min(Area_1, Area_2)}{\max(Area_1, Area_2)} > 0.45 \tag{2.20}$$
$$1.05 \leq UnionRatio = \frac{UnionWidth}{UnionHeight} \leq 2.20 \tag{2.21}$$
$$HorizontalDistance < 0.65 \times UnionWidth \tag{2.22}$$
$$VerticalDistance < 0.30 \times UnionHeight \tag{2.23}$$

Case 2: Two Hands Forming a Single Entity The combined area and shape should satisfy:

$$Area \geq 5500 \tag{2.24}$$
$$HeightWidthRatio > 1.20, \quad WidthHeightRatio < 0.90 \tag{2.25}$$

If the above is not satisfied, apply relaxed criteria:

$$1.35 < WidthHeightRatio \leq 2.30 \tag{2.26}$$
$$0.42 \leq Extent \leq 0.75 \tag{2.27}$$
$$0.55 \leq Solidity \leq 0.86 \tag{2.28}$$
$$DefectsCount \geq 2 \tag{2.29}$$
$$MaxDepthRatio \geq 0.11 \tag{2.30}$$

These quantitative rules ensure the gesture has sufficient area, moderate compactness, and at least two clear depressions. The complete gesture classification logic is implemented in Listing 6.

UI Interface and Page Logic The projection interface is implemented as a full-screen Tkinter application. The home page contains two interactive cards: the clock card and the weather card. When the system detects that the clock area is blocked, it switches to the time page displaying current time, date, and working day. When the weather area is blocked, it switches to the weather page displaying current weather information. A status label provides real-time feedback on the recognition results. The full implementation of the Tkinter-based UI system is shown in Listing 3.

The UI module contains the home page, time page, weather page, gesture status updates, and automatic return to home functionality which is shown in Fig. 4. The time page updates every second via Tkinter's timing mechanism, and a thread-safe UI update function ensures safe interface updates from the gesture recognition thread.

Figure 4: Home, Time, and Weather Pages of the Projection UI

Occlusion-Based Icon Interaction In addition to gesture recognition, the system supports interaction via occluding projected icons. The clock and weather icons define two fixed ROIs. During initialization, baseline brightness is estimated from multiple frames. If current brightness in an ROI drops below the predetermined threshold, the system triggers the corresponding page: clock ROI triggers the time page, weather ROI triggers the weather page. The brightness-based occlusion detection algorithm is implemented in Listing 5.

3 Verification

3.1 Mechanical Module

For the mechanical part, the verification mainly checked whether the structure could support the whole system during normal operation. After assembly, I checked that the printed holders could carry the projector, camera, speaker, and power bank without obvious bending or sliding. We also let the robot move on the desk in several basic directions and observed whether the upper structure shook too much, whether screws became loose, and whether cables touched the wheels. Since the project did not require high-speed motion, this verification was mainly about making sure the robot could move, stop, and turn while keeping the sensing and projection modules in place.

During testing, I made several small adjustments. Some early printed parts were too tight for the real modules, and some cable positions made the top platform messy or blocked later assembly. I modified the supports to give more clearance, fixed the main modules more directly, and routed cables away from the wheels and the camera view. The final assembled prototype used for this check is shown in Fig. 5. After these changes, the prototype could complete basic movement and function testing without the frame falling apart or the devices shifting. This result showed that the mechanical design was acceptable for the final course prototype, although a cleaner enclosure and stronger integrated mounting would still be needed for a more polished version.

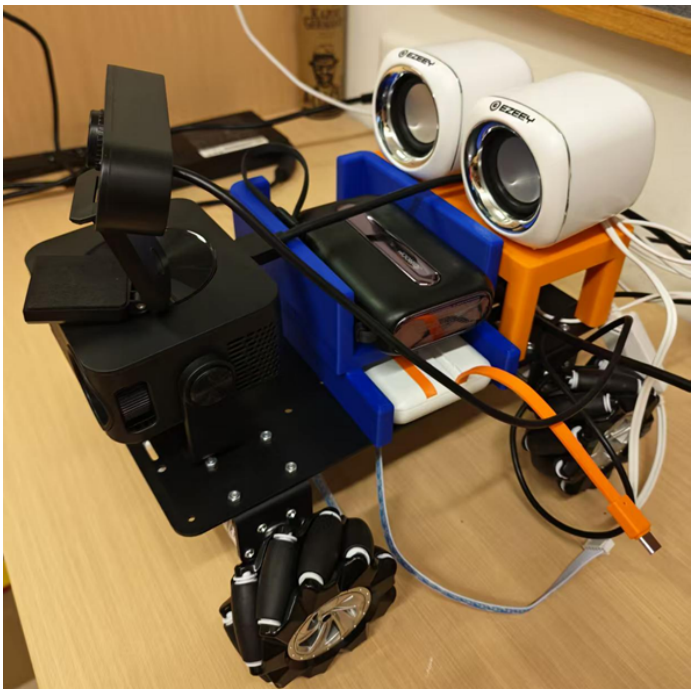


Figure 5: Assembled prototype with the purchased chassis and the 3D-printed upper support structure installed.

3.2 Motor Control Module

The original design goal for the Mecanum wheel module was to enable omnidirectional movement, precise positioning, and responsive motion control of the robot. Due to budget and hardware constraints, the module was only implemented with **direct power supply driving the wheels**.

3.2.1 Verification Procedures and Results

1. **Wheel Rotation Verification**
    - **Method:** Apply power and observe if all four wheels rotate properly.
    - **Result:** All wheels rotate smoothly, without unusual noise or stalling.
    - **Conclusion:** Hardware connections and basic mechanical installation are reliable.
2. **Mechanical Installation Stability Verification**
    - **Method:** Push the robot lightly on a flat surface and observe wheel and chassis stability.
    - **Result:** Wheels and motors remain firmly attached; the structure is stable.
    - **Conclusion:** The module is structurally safe and capable of demonstration-level operation.
3. **Theoretical Design Metrics**
    - Original targets: maximum linear speed ≥ 0.1 m/s, positioning accuracy $\leq \pm 2$ cm, command response delay ≤ 500 ms.
    - Actual implementation: wheel rotation is possible, command response delay ≤ 500 ms.
4. **PWM-based Mecanum-wheel Design**
    - To verify the PWM-based mecanum-wheel design, each wheel should first be tested independently at several duty cycles, such as 30%, 60%, and 90%, to confirm that the wheel speed changes consistently with the PWM input. The direction pins should also be tested to confirm forward and reverse rotation for each motor. After individual wheel testing, the full robot should be tested for forward and backward movement, left and right lateral movement, diagonal movement, and rotation in place.
    - For each test, the expected wheel-direction pattern and the actual robot movement should be recorded. A successful result would show that the robot moves in the commanded direction without obvious wheel conflict, slipping, or unstable shaking. Position accuracy could be checked by commanding the robot to move a fixed distance on the desk and measuring the displacement error. These

tests would verify that the PWM signals are not only rotating the wheels, but also producing the intended mecanum-wheel motion of the whole device.

3.3 Software Interaction and Gesture Recognition Module

The software verification focuses on whether the visual recognition results correctly trigger the corresponding UI updates and audio feedback. Verification is divided into three parts: gesture recognition verification, interface page switching verification, and response delay verification which is shown in Table 2.

For gesture recognition, three gestures were tested: scissors, thumbs-up, and double-heart. Each gesture was executed multiple times within the designated ROI. If the system correctly recognized the gesture and updated the status label or triggered audio feedback, the attempt was considered successful.

Interface page switching was verified by manually blocking the clock and weather icon areas. Successful attempts required the interface to switch correctly to the corresponding pages and display the correct time and weather information.

Response delay was measured as the time between user action and visible UI/audio response. If the delay was short enough for real-time interaction and the debounce mechanism prevented repeated triggering, the attempt was considered successful. The system status module also verified that non-home pages returned automatically to the main screen after a preset timeout.

Test Item	Method	Success Criterion	Result
Palm gesture recognition	Perform palm gesture 20 times inside gesture ROI	Accuracy $\geq$ 85%	Satisfied
Fist gesture recognition	Perform fist gesture 20 times inside gesture ROI	Accuracy $\geq$ 85%	Satisfied
Two-finger gesture recognition	Perform two-finger gesture 20 times inside gesture ROI	Accuracy $\geq$ 85%	Satisfied
Clock page switching	Occlude clock icon region 20 times	Correctly switches to time page	Satisfied
Weather page switching	Occlude weather icon region 20 times	Correctly switches to weather page	Satisfied
UI response latency	Measure action-to-display delay	Delay $<$ 1 s	Satisfied
Auto return home	Stay on time/weather page without action	Returns after about 6 s	Satisfied
False trigger test	No gesture/occlusion for 2 minutes	No unexpected trigger	Satisfied

Table 2: Verification results for software interaction and gesture recognition

3.4 System Verification

3.4.1 User-Friendly Design Requirements for a Motion-Capable Robotic Assistant

In addition to functional performance, a motion-capable robotic assistant should satisfy several user-friendly design requirements to ensure practical usability in real-world applications. These requirements include intuitive interaction, real-time responsiveness, motion stability, environmental adaptability, and accessibility for users with different levels of technical experience.

First, intuitive human–robot interaction is an essential requirement for improving user experience. Users should be able to interact with the robotic assistant naturally without requiring complicated training procedures or specialized operational knowledge. In this project, gesture recognition is adopted as the primary interaction method, allowing users to issue commands through simple and familiar hand gestures. This interaction approach reduces operational complexity and improves the accessibility of the system.

Real-time responsiveness is another important requirement for a motion-capable robotic assistant. Delayed responses between user commands and robot actions may negatively affect usability and interaction continuity. Therefore, the system is designed with lightweight image processing and efficient gesture classification algorithms to minimize recognition latency and ensure smooth real-time interaction between the user and the robotic platform.

Motion stability and operational safety are also critical requirements. Since the robotic assistant performs physical movement during operation, the system must maintain stable motion behavior while accurately responding to user instructions. Stable movement improves user confidence in the system and reduces the risk of unexpected motion during operation. In addition, the robotic assistant should be capable of operating reliably under different environmental conditions, including variations in lighting and background complexity. To improve robustness, HSV-based image segmentation and contour analysis are used to enhance gesture detection performance under varying visual conditions.

Finally, accessibility and ease of use are emphasized in the overall system design. The interface and control workflow are simplified so that users with limited technical backgrounds can operate the robotic assistant effectively. By combining intuitive gesture-based interaction with real-time motion control, the proposed system aims to provide a practical and user-friendly robotic assistant for daily human–robot interaction scenarios.

4 Costs

This section summarizes the estimated costs of the project, including labor costs for each team member and the material costs of components used in the system.

4.1 Labor Cost Calculation

The labor cost for each team member shown in Table 3 is estimated using the formula:

$$\text{Labor Cost} = \text{Hourly Rate} \times \text{Hours Worked} \times 2.5$$

Name	**Task Description**	**Estimated Hours**	**Hourly Rate (CNY)**	**Labor Cost (CNY)**
Ziyang Jin	Gesture Recognition & UI Development	80	50	10,000
Cheng Zheng	Raspberry Pi Configuration & System Flashing	60	50	7,500
Zhewei Zhang	Vehicle Structure and Assembly	70	50	8,750
Yuxuan Wu	Mecanum Wheel Drive Implementation & Testing	50	50	6,250

Table 3: Estimated labor cost for each team member

4.2 Material Cost

The Table 4 summarizes the cost of components used in the project, including purchased and development-consumed materials:

No.	Name	Price (CNY)	Notes
1	Raspberry Pi 4 Model B, 4GB	938.5	Main controller
2	STM32 4-Wheel Electric Car	290	Platform
3	Projector	188	Projection screen
4	Camera	28.88	Gesture and occlusion capture
5	USB Hub	29.9	Raspberry Pi USB expansion
6	SD Card 64GB	89	Storage
7	Speaker	30	Audio output
8	Micro-HDMI to HDMI Cable	23.9	Raspberry Pi to projector
9	Push-button Switch	7.7	Raspberry Pi physical switch
10	502 Glue	11.1	Equipment fixing
Development-consumed materials			
11	SD Card 32GB	23.94	Counterfeit, unreadable
12	SD Card 64GB	74.9	Damaged during use
13	Audio Playback Module	29	Interface incompatible
14	11200mAh 5V Li-ion Battery	58.44	Interface incompatible

Table 4: Material cost of components

Additional Notes: Materials provided by the team and not purchased include 3D printed mounts, one 10,000 mAh integrated main battery with voltage regulation and conversion circuits, SD card readers, and USB keyboards.

5 Conclusions

In this project, we built a portable robot prototype that combines a projected interface, camera-based gesture recognition, basic movement, and sound feedback. The robot can display simple information such as weather and time, and it can recognize different hand gestures to trigger responses such as happy or sad sounds. The final result showed that these separate modules could be integrated into one working platform. It also showed that projection can be a feasible way to make a small robot more interactive without adding a traditional screen, although the current system is still closer to a lab prototype than a finished product.

There are still several limitations. The mechanical structure was not fully custom-made, because we finally used a purchased chassis for better reliability instead of continuing with the weaker self-designed base. Gesture recognition and projection quality can also be affected by lighting, camera angle, and cable placement. From an ethics and safety point of view, the camera should only be used for real-time recognition and should not store user images without permission. The projector should not be aimed directly at users' eyes, and the moving robot should be tested at safe speeds. If we had more time, the next improvements would be a more integrated enclosure, cleaner cable management, better camera-projector calibration, and a smaller and more stable overall structure.

Future work: In future research, the current gesture recognition method based on OpenCV can be expanded into a recognition system based on neural networks. Firstly, the hardware platform should be upgraded to meet the computational requirements for neural network inference, such as using more powerful embedded boards or adding artificial intelligence acceleration modules. This will enable the system to run lightweight deep learning models in real time.

Second, a large labeled gesture dataset should be collected to support model training. This dataset should include various users, lighting conditions, hand positions, backgrounds, and projection environments. Each image or video sample should be labeled with the corresponding gesture or interaction command. To reduce the difficulty of training from scratch, pretrained models can also be used as the starting point. Based on the system requirements, lightweight models such as MobileNet can be considered for gesture classification, while U-Net-like structures can be considered for hand-region segmentation or interaction-area detection.

In addition, OpenCV does not need to be completely removed. Some OpenCV-based features and rule-based results can be retained and combined with neural network predictions through weighted fusion. In this way, the system can retain some of the explainability of the current method while improving the recognition accuracy and robustness. This future upgrade will make the interactive system more flexible and enable it to support more gesture types and user interaction functions.

Appendix A Source Code

A.1 Main Program

Listing 1: main.py: Main control program and vision-processing thread

```
import threading
import time
import cv2

from config import DEBUG_CAMERA, OCCLUSION_COOLDOWN, GESTURE_COOLDOWN,
    AUTO_BACK_SECONDS
from ui.projector_ui import ProjectorUI
from vision.camera import Camera
from vision.occlusion_detector import OcclusionDetector
from vision.gesture_detector import GestureDetector
from audio.audio_player import AudioPlayer
from utils.state import AppState
from utils.logger import info, warn, error

def vision_loop(ui):
    camera = Camera()
    occlusion = OcclusionDetector()
    gesture = GestureDetector()
    audio = AudioPlayer()
    state = AppState()

    info("System on")
    ui.call(lambda: ui.set_status("Camera On"))

    while True:
        frame = camera.read()

        if frame is None:
            warn("Camera Offline")
            ui.call(lambda: ui.set_status("Camera failed, check USB
                link"))
            time.sleep(0.2)
            continue

        event = occlusion.detect(frame)

        if event and state.can_trigger_occlusion(OCCLUSION_COOLDOWN):
            if event == "clock":
                state.mark_occlusion("time")
                ui.call(ui.show_time)
                info("Time mode is activated")
            elif event == "weather":
```

```
                state.mark_occlusion("weather")
                ui.call(ui.show_weather)
                info("Weather mode is activated")

        g = gesture.detect(frame)

        if state.can_trigger_gesture(g, GESTURE_COOLDOWN):
            audio.play(g)
            state.mark_gesture(g)
            ui.call(lambda gesture_name=g: ui.show_gesture_status(
               gesture_name))
            info(f"Find gesture {g}")

        if state.should_back_home(AUTO_BACK_SECONDS):
            state.back_home()
            ui.call(ui.back_home)
            info("Back to home page")

        if DEBUG_CAMERA:
            show = frame.copy()
            occlusion.draw_debug(show)
            gesture.draw_debug(show)

            cv2.putText(
                show,
                f"Page: {state.page}",
                (20, 34),
                cv2.FONT_HERSHEY_SIMPLEX,
                0.8,
                (0, 255, 255),
                2
            )

            cv2.imshow("Camera Debug", show)

            if cv2.waitKey(1) & 0xFF == ord("q"):
                break

    camera.release()
    cv2.destroyAllWindows()
    info("Exit System")

def main():
    ui = ProjectorUI()
    t = threading.Thread(target=vision_loop, args=(ui,), daemon=True)
    t.start()
    ui.run()
```

```
87
88 if __name__ == "__main__":
89     main()
```

A.2 System Configuration

Listing 2: config.py: System configuration parameters

```
1  CAMERA_INDEX = 0
2  FRAME_WIDTH = 640
3  FRAME_HEIGHT = 480
4
5  DEBUG_CAMERA = True
6
7  OCCLUSION_BASELINE_FRAMES = 50
8  OCCLUSION_THRESHOLD_RATIO = 0.70
9  OCCLUSION_CONFIRM_FRAMES = 8
10 OCCLUSION_COOLDOWN = 4.0
11
12 CLOCK_ROI = (0.10, 0.25, 0.42, 0.75)
13 WEATHER_ROI = (0.58, 0.25, 0.90, 0.75)
14
15 GESTURE_ROI = (0.25, 0.12, 0.75, 0.88)
16 GESTURE_CONFIRM_FRAMES = 15
17 GESTURE_COOLDOWN = 4.0
18
19 AUTO_BACK_SECONDS = 6
20
21 AUDIO_1 = "audio/files/gesture_1.mp3"
22 AUDIO_2 = "audio/files/gesture_2.mp3"
23 AUDIO_3 = "audio/files/gesture_3.mp3"
24
25 CITY_NAME = "      "
26 WEATHER_TEXT = "           \ n 2 4  \   n     65%        "
```

A.3 User Interface Module

Listing 3: projector_ui.py: Projection UI and page logic

```
1 import tkinter as tk
2 from services.time_service import get_current_time, get_current_date,
     get_weekday
3 from services.weather_service import get_weather
4
5 class ProjectorUI:
6     def __init__(self):
7         self.root = tk.Tk()
```

```
        self.root.title("Smart Projection System")
        self.root.attributes("-fullscreen", True)
        self.root.configure(bg="#07111f")
        self.page = None
        self.status_text = "System Start"

        self.container = tk.Frame(self.root, bg="#07111f")
        self.container.pack(fill="both", expand=True)

        self.status_label = tk.Label(
            self.root,
            text=self.status_text,
            font=("Arial", 18),
            fg="#22c55e",
            bg="#07111f"
        )
        self.status_label.place(relx=0.5, rely=0.95, anchor="center")

        self.root.bind("<Escape>", lambda e: self.root.destroy())
        self.show_home()

    def clear(self):
        for widget in self.container.winfo_children():
            widget.destroy()

    def set_status(self, text):
        self.status_text = text
        self.status_label.config(text=text)

    def header(self, title):
        tk.Label(
            self.container,
            text=title,
            font=("Arial", 48, "bold"),
            fg="#f8fafc",
            bg="#07111f"
        ).pack(pady=(42, 8))


    def card(self, parent, icon, title, desc, color):
        box = tk.Frame(parent, bg="#0f1f35", width=440, height=320)
        box.pack_propagate(False)

        tk.Label(
            box,
            text=icon,
            font=("Arial", 76),
```

```
            fg=color,
            bg="#0f1f35"
        ).pack(pady=(38, 10))

        tk.Label(
            box,
            text=title,
            font=("Arial", 32, "bold"),
            fg="#f8fafc",
            bg="#0f1f35"
        ).pack()

        tk.Label(
            box,
            text=desc,
            font=("Arial", 18),
            fg="#cbd5e1",
            bg="#0f1f35",
            wraplength=350,
            justify="center"
        ).pack(pady=(16, 0))

        return box

    def show_home(self):
        self.page = "home"
        self.clear()

        self.header(
            "Smart Projection System",

        )

        cards = tk.Frame(self.container, bg="#07111f")
        cards.pack(expand=True)

        left = self.card(
            cards,
            "    ",
            "Time Interaction",
            "Switch to Time Page",
            "#38bdf8"
        )
        left.grid(row=0, column=0, padx=42)

        right = self.card(
            cards,
```

```
            "   ",
            "Weather Interaction",
            "Switch to Weather Interaction",
            "#facc15"
        )
        right.grid(row=0, column=1, padx=42)

        self.set_status("System is ready")

    def show_time(self):
        self.page = "time"
        self.clear()

        self.header("Current Time")

        self.time_label = tk.Label(
            self.container,
            text=get_current_time(),
            font=("Arial", 108, "bold"),
            fg="#38bdf8",
            bg="#07111f"
        )
        self.time_label.pack(expand=True)

        tk.Label(
            self.container,
            text=f"{get_current_date()}  {get_weekday()}",
            font=("Arial", 34),
            fg="#e2e8f0",
            bg="#07111f"
        ).pack(pady=(0, 70))

        self.set_status("Time module is activated")
        self.refresh_time()

    def refresh_time(self):
        if self.page == "time":
            self.time_label.config(text=get_current_time())
            self.root.after(1000, self.refresh_time)

    def show_weather(self):
        self.page = "weather"
        self.clear()

        self.header("Current Weather")

        tk.Label(
```

```
            self.container,
            text="    ",
            font=("Arial", 118),
            fg="#facc15",
            bg="#07111f"
        ).pack(pady=(34, 0))

        tk.Label(
            self.container,
            text=get_weather(),
            font=("Arial", 56, "bold"),
            fg="#f8fafc",
            bg="#07111f",
            wraplength=920,
            justify="center"
        ).pack(expand=True)

        self.set_status("Weather Module is activated")

    def show_gesture_status(self, gesture):
        name = {
            "thumb": "thumb",
            "double_heart": "Heart",
            "two": "victory sign"
        }.get(gesture, gesture)

        self.set_status(f" Find Gesture {name}  audio   is played")

    def back_home(self):
        self.show_home()

    def run(self):
        self.root.mainloop()

    def call(self, func):
        self.root.after(0, func)
```

A.4 Vision Modules

Listing 4: camera.py: Camera initialization and frame acquisition

```
import cv2

class Camera:
    def __init__(self, index=0, width=640, height=480):
        self.cap = cv2.VideoCapture(index)
        self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, width)
```

```
        self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, height)

    def read(self):
        if not self.cap.isOpened():
            return None
        ret, frame = self.cap.read()
        if not ret:
            return None
        return frame

    def release(self):
        if self.cap:
            self.cap.release()
```

Listing 5: occlusion_detector.py: Brightness-based icon occlusion detection

```
import cv2
import numpy as np
from config import CLOCK_ROI, WEATHER_ROI, OCCLUSION_BASELINE_FRAMES,
    OCCLUSION_THRESHOLD_RATIO

class OcclusionDetector:
    def __init__(self):
        self.frame_count = 0
        self.clock_vals = []
        self.weather_vals = []
        self.clock_base = None
        self.weather_base = None

    def _roi(self, frame, roi):
        h, w = frame.shape[:2]
        x1, y1, x2, y2 = roi
        return frame[int(y1*h):int(y2*h), int(x1*w):int(x2*w)]

    def _brightness(self, img):
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        return float(np.mean(gray))

    def detect(self, frame):
        clock_img = self._roi(frame, CLOCK_ROI)
        weather_img = self._roi(frame, WEATHER_ROI)

        clock_v = self._brightness(clock_img)
        weather_v = self._brightness(weather_img)

        if self.frame_count < OCCLUSION_BASELINE_FRAMES:
            self.clock_vals.append(clock_v)
            self.weather_vals.append(weather_v)
```

```
            self.frame_count += 1

            if self.frame_count == OCCLUSION_BASELINE_FRAMES:
                self.clock_base = float(np.mean(self.clock_vals))
                self.weather_base = float(np.mean(self.weather_vals))

            return None

        if self.clock_base and clock_v < self.clock_base *
           OCCLUSION_THRESHOLD_RATIO:
            return "clock"

        if self.weather_base and weather_v < self.weather_base *
           OCCLUSION_THRESHOLD_RATIO:
            return "weather"

        return None

    def draw_debug(self, frame):
        h, w = frame.shape[:2]

        def draw(roi, name, color):
            x1, y1, x2, y2 = roi
            p1 = (int(x1*w), int(y1*h))
            p2 = (int(x2*w), int(y2*h))
            cv2.rectangle(frame, p1, p2, color, 2)
            cv2.putText(frame, name, (p1[0], p1[1]-8),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)

        draw(CLOCK_ROI, "CLOCK", (0, 255, 255))
        draw(WEATHER_ROI, "WEATHER", (255, 255, 0))

        return frame
```

Listing 6: gesture_detector.py: Gesture recognition using skin segmentation and contour analysis

```
import cv2
import numpy as np
import math
from config import GESTURE_ROI


class GestureDetector:
    def __init__(self):
        # Kernel for noise removal
        self.kernel = np.ones((5, 5), np.uint8)

```

```
    def _roi(self, frame):
        # Get gesture detection area
        h, w = frame.shape[:2]
        x1, y1, x2, y2 = GESTURE_ROI

        roi = frame[int(y1*h):int(y2*h), int(x1*w):int(x2*w)]

        return roi, (
            int(x1*w),
            int(y1*h),
            int(x2*w),
            int(y2*h)
        )

    def _skin_mask(self, roi):
        # Convert image to HSV color space
        hsv = cv2.cvtColor(roi, cv2.COLOR_BGR2HSV)

        # Skin color range
        lower1 = np.array([0, 20, 30])
        upper1 = np.array([30, 255, 255])

        lower2 = np.array([150, 20, 30])
        upper2 = np.array([180, 255, 255])

        mask1 = cv2.inRange(hsv, lower1, upper1)
        mask2 = cv2.inRange(hsv, lower2, upper2)
        mask = cv2.bitwise_or(mask1, mask2)

        # Remove noise and fill small holes
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, self.kernel)
        mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, self.kernel)

        return mask

    def _defect_features(self, contour, hull_index, w, h):
        # Get convexity defect features
        if hull_index is None or len(hull_index) <= 3:
            return 0, 0.0, 180.0

        defects = cv2.convexityDefects(contour, hull_index)

        if defects is None:
            return 0, 0.0, 180.0

        defects_count = 0
        max_depth = 0.0
```

```
        min_angle = 180.0

        min_depth = max(w, h) * 0.055

        for i in range(defects.shape[0]):
            s, e, f, d = defects[i, 0]

            start = contour[s][0]
            end = contour[e][0]
            far = contour[f][0]

            a = np.linalg.norm(end - start)
            b = np.linalg.norm(far - start)
            c = np.linalg.norm(end - far)

            if b * c == 0:
                continue

            cos_value = (b*b + c*c - a*a) / (2*b*c)
            cos_value = np.clip(cos_value, -1.0, 1.0)

            angle = math.acos(cos_value)
            angle = angle * 180 / math.pi

            depth = d / 256.0
            max_depth = max(max_depth, depth)

            if depth > min_depth and angle < 95:
                defects_count += 1
                min_angle = min(min_angle, angle)

        max_depth_ratio = max_depth / max(max(w, h), 1)

        return defects_count, max_depth_ratio, min_angle

    def _count_defects(self, contour, hull_index, w, h):
        # Only return defect count
        defects_count, _, _ = self._defect_features(contour, hull_index
            , w, h)
        return defects_count

    def _detect_double_heart(self, contours):
        # Detect double-hand heart gesture
        valid = []

        for c in contours:
            area = cv2.contourArea(c)
```

```
105
106             if area > 1200:
107                 valid.append(c)
108
109         if len(valid) == 0:
110             return False
111
112         valid = sorted(valid, key=cv2.contourArea, reverse=True)
113
114         # Case 1: two hands are separated
115         if len(valid) >= 2:
116             c1, c2 = valid[0], valid[1]
117
118             x1, y1, w1, h1 = cv2.boundingRect(c1)
119             x2, y2, w2, h2 = cv2.boundingRect(c2)
120
121             area1 = cv2.contourArea(c1)
122             area2 = cv2.contourArea(c2)
123
124             area_ratio = min(area1, area2) / max(area1, area2)
125
126             center1 = (x1 + w1 / 2, y1 + h1 / 2)
127             center2 = (x2 + w2 / 2, y2 + h2 / 2)
128
129             left = min(x1, x2)
130             right = max(x1 + w1, x2 + w2)
131             top = min(y1, y2)
132             bottom = max(y1 + h1, y2 + h2)
133
134             union_w = right - left
135             union_h = bottom - top
136
137             union_ratio = union_w / max(union_h, 1)
138             horizontal_distance = abs(center1[0] - center2[0])
139             vertical_distance = abs(center1[1] - center2[1])
140
141             # Two hands should be similar in size and close to each
                    other
142             if (
143                 area_ratio > 0.30 and
144                 0.90 <= union_ratio <= 2.60 and
145                 horizontal_distance < union_w * 0.80 and
146                 vertical_distance < union_h * 0.40
147             ):
148                 return True
149
150         # Case 2: two hands are connected as one contour
```

```
        c = valid[0]
        area = cv2.contourArea(c)

        if area < 5500:
            return False

        hull = cv2.convexHull(c)
        hull_area = cv2.contourArea(hull)

        if hull_area <= 0:
            return False

        solidity = area / hull_area

        x, y, w, h = cv2.boundingRect(c)

        width_height_ratio = w / max(h, 1)
        height_width_ratio = h / max(w, 1)
        extent = area / max(w * h, 1)

        hull_index = cv2.convexHull(c, returnPoints=False)

        defects_count, max_depth_ratio, min_angle = self.
           _defect_features(
            c,
            hull_index,
            w,
            h
        )

        # Exclude obvious thumb-up shape
        if height_width_ratio > 1.20 and width_height_ratio < 0.90:
            return False

        # Connected double heart is usually wider than thumb-up
        if (
            width_height_ratio > 1.15 and
            width_height_ratio <= 2.60 and
            0.38 <= extent <= 0.82 and
            0.50 <= solidity <= 0.90 and
            defects_count >= 1 and
            max_depth_ratio >= 0.08
        ):
            return True

        return False

```

```
    def detect(self, frame):
        roi, _ = self._roi(frame)
        mask = self._skin_mask(roi)

        contours, _ = cv2.findContours(
            mask,
            cv2.RETR_EXTERNAL,
            cv2.CHAIN_APPROX_SIMPLE
        )

        if not contours:
            return None

        # Gesture 1: double-hand heart
        if self._detect_double_heart(contours):
            return "double_heart"

        contour = max(contours, key=cv2.contourArea)
        area = cv2.contourArea(contour)

        if area < 3000:
            return None

        hull = cv2.convexHull(contour)
        hull_area = cv2.contourArea(hull)

        if hull_area <= 0:
            return None

        solidity = area / hull_area

        x, y, w, h = cv2.boundingRect(contour)

        ratio = h / max(w, 1)
        extent = area / max(w * h, 1)

        hull_index = cv2.convexHull(contour, returnPoints=False)

        defects_count, max_depth_ratio, min_angle = self.
           _defect_features(
            contour,
            hull_index,
            w,
            h
        )

        # Debug information
```

```
243             print(
244                 "area:", int(area),
245                 "solidity:", round(solidity, 2),
246                 "ratio:", round(ratio, 2),
247                 "extent:", round(extent, 2),
248                 "defects:", defects_count,
249                 "max_depth_ratio:", round(max_depth_ratio, 2),
250                 "min_angle:", round(min_angle, 1)
251             )
252 
253             # Gesture 2: thumb up
254             # Thumb is tall, compact, and has shallow defects
255             if (
256                 ratio > 1.15 and
257                 defects_count <= 1 and
258                 solidity > 0.70 and
259                 extent > 0.30 and
260                 max_depth_ratio < 0.18
261             ):
262                 return "thumb"
263 
264             # Gesture 3: scissors
265             # Scissors has a deeper and sharper gap between fingers
266             if (
267                 ratio > 1.05 and
268                 defects_count >= 1 and
269                 solidity < 0.92 and
270                 max_depth_ratio >= 0.18 and
271                 min_angle < 80
272             ):
273                 return "two"
274 
275             return None
276 
277         def draw_debug(self, frame):
278             # Draw gesture detection area
279             h, w = frame.shape[:2]
280             x1, y1, x2, y2 = GESTURE_ROI
281 
282             p1 = (int(x1*w), int(y1*h))
283             p2 = (int(x2*w), int(y2*h))
284 
285             cv2.rectangle(frame, p1, p2, (0, 255, 0), 2)
286 
287             cv2.putText(
288                 frame,
289                 "GESTURE",
```

```
            (p1[0], p1[1] - 8),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.7,
            (0, 255, 0),
            2
        )

        return frame
```

A.5 Audio Module

Listing 7: audio_player.py: Gesture-triggered audio playback

```
import os
import pygame
from config import AUDIO_1, AUDIO_2, AUDIO_3

class AudioPlayer:
    def __init__(self):
        pygame.mixer.init()
        self.map = {
            "thumb": AUDIO_1,
            "double_heart": AUDIO_2,
            "two": AUDIO_3
        }

    def play(self, gesture):
        path = self.map.get(gesture)
        if not path:
            return

        path = os.path.join(os.getcwd(), path)

        if not os.path.exists(path):
            print(f"Audio not found: {path}")
            return

        try:
            if pygame.mixer.music.get_busy():
                pygame.mixer.music.stop()

            pygame.mixer.music.load(path)
            pygame.mixer.music.play()

        except Exception as e:
            print(f"Broadcast Fail: {e}")
```

A.6 Service Modules

Listing 8: time_service.py: Time and date service

```python
import datetime

def get_current_time():
    return datetime.datetime.now().strftime("%H:%M:%S")

def get_current_date():
    return datetime.datetime.now().strftime("%Y-%m-%d")

def get_weekday():
    weekdays = [
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    ]
    return weekdays[datetime.datetime.now().weekday()]

def get_full_time_text():
    return f"{get_current_date()} {get_weekday()} {get_current_time()}"

def get_hour_minute():
    return datetime.datetime.now().strftime("%H:%M")

def get_second():
    return datetime.datetime.now().strftime("%S")
```

Listing 9: weather_service.py: Weather data request and cache service

```python
import json
import os
import time
import urllib.parse
import urllib.request

CACHE_FILE = "weather_cache.json"
CACHE_SECONDS = 600

DEFAULT_WEATHER = {
    "city": "Haining",
    "weather": "Sunny",
    "temperature": "24",
    "wind": "Breeze",
```

```
    "humidity": "65"
}

def _read_cache():
    if not os.path.exists(CACHE_FILE):
        return None

    try:
        with open(CACHE_FILE, "r", encoding="utf-8") as f:
            data = json.load(f)

        if time.time() - data.get("time", 0) <= CACHE_SECONDS:
            return data.get("weather")

        return None
    except Exception:
        return None

def _write_cache(weather):
    try:
        with open(CACHE_FILE, "w", encoding="utf-8") as f:
            json.dump({
                "time": time.time(),
                "weather": weather
            }, f, ensure_ascii=False, indent=2)
    except Exception:
        pass

def _request_weather(city="Jiaxing"):
    url = "https://wttr.in/" + urllib.parse.quote(city) + "?format=j1"

    req = urllib.request.Request(
        url,
        headers={
            "User-Agent": "Mozilla/5.0"
        }
    )

    with urllib.request.urlopen(req, timeout=5) as response:
        raw = response.read().decode("utf-8")
        data = json.loads(raw)

    current = data.get("current_condition", [{}])[0]

    weather_desc = current.get("lang_zh", [{}])[0].get("value") or
       current.get("weatherDesc", [{}])[0].get("value", "    ")

```

```
    return {
        "city": city,
        "weather": weather_desc,
        "temperature": current.get("temp_C", "24"),
        "wind": current.get("windspeedKmph", "0") + "km/h",
        "humidity": current.get("humidity", "65")
    }

def get_weather_data(city="Jiaxiang"):
    cache = _read_cache()
    if cache:
        return cache

    try:
        weather = _request_weather(city)
        _write_cache(weather)
        return weather
    except Exception:
        return DEFAULT_WEATHER

def get_weather(city="Jiaxing"):
    data = get_weather_data(city)

    return (
        f"{data['city']}  {data['weather']}\n"
        f"{data['temperature']}   \n"
        f"Humidity {data['humidity']}%  Wind Speed {data['wind']}"
    )
```

A.7 Utility Modules

Listing 10: state.py: Application state and cooldown management

```
import time

class AppState:
    def __init__(self):
        self.page = "home"
        self.last_occlusion_time = 0
        self.last_gesture_time = 0
        self.last_gesture = None
        self.status_text = "System Starting"
        self.last_page_change_time = time.time()

    def can_trigger_occlusion(self, cooldown):
        return time.time() - self.last_occlusion_time >= cooldown

```

```
    def mark_occlusion(self, page):
        self.page = page
        self.last_occlusion_time = time.time()
        self.last_page_change_time = time.time()

    def can_trigger_gesture(self, gesture, cooldown):
        if not gesture:
            return False
        if gesture != self.last_gesture:
            return True
        return time.time() - self.last_gesture_time >= cooldown

    def mark_gesture(self, gesture):
        self.last_gesture = gesture
        self.last_gesture_time = time.time()

    def should_back_home(self, seconds):
        if self.page == "home":
            return False
        return time.time() - self.last_page_change_time >= seconds

    def back_home(self):
        self.page = "home"
        self.last_page_change_time = time.time()

    def set_status(self, text):
        self.status_text = text
```

Listing 11: logger.py: Runtime logging utility

```
import datetime
import os

LOG_FILE = "app.log"

def _time():
    return datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

def _write(level, text):
    line = f"[{_time()}] [{level}] {text}"
    print(line)
    try:
        with open(LOG_FILE, "a", encoding="utf-8") as f:
            f.write(line + "\n")
    except Exception:
        pass

def info(text):
```

```
19      _write("INFO", text)
20
21  def warn(text):
22      _write("WARN", text)
23
24  def error(text):
25      _write("ERROR", text)
26
27  def debug(text):
28      _write("DEBUG", text)
```