Универзитет „Св. Кирил и Методиј" во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Course:

DIGITAL IMAGE PROCESSING

Seminar Paper

Topic:

Gesture Recognition Game Controller

Mentor:

Ivica Dimitrovski Ph.D

Students:

Edon Fetaji, 221517

Leon Saraqini, 211508

TABLE OF CONTENTS

1. Introduction

Gesture recognition is a key topic in digital image processing and computer vision, where the objective is to interpret human motion and translate it into commands that software systems can understand. This project implements a real-time hand gesture recognition system that controls a desktop application and can optionally control external games through keyboard emulation.

*This seminar project implements a complete real-time gesture recognition system: not only a classifier, but a **usable controller** integrated into a desktop application. The final application is designed as a "gesture-driven menu + gesture-to-keyboard controller." The user interacts with the GUI entirely via gestures, and can also use gestures to trigger keyboard commands mapped to games such as Subway Surfers and Temple Run.*

The system captures video frames from a webcam, detects and tracks a hand, recognizes the performed gesture, and translates it into a high-level action (e.g., LEFT, RIGHT, JUMP, DUCK, SELECT). These actions are used to navigate a menu-driven graphical user interface (GUI) and to send keyboard inputs to games such as Subway Surfers or Temple Run, depending on the selected game profile.

A practical gesture controller must be stable and predictable. Computer vision models can produce noisy per-frame outputs due to illumination changes, motion blur, occlusions, and background clutter. For that reason, this project emphasizes temporal filtering, debouncing, and cooldown logic to ensure that user actions occur at an appropriate rate and that the UI remains responsive.

*Gesture classifiers typically output results **per frame**, meaning a gesture may be detected repeatedly across many consecutive frames. Without additional logic, this produces repeated actions (e.g., menu moving too fast, repeated selection triggers, repeated key presses in a game). Therefore, the project emphasizes "vision → stable discrete control*

Github Repository Link with the code for the project implementation

PowerPoint presentation

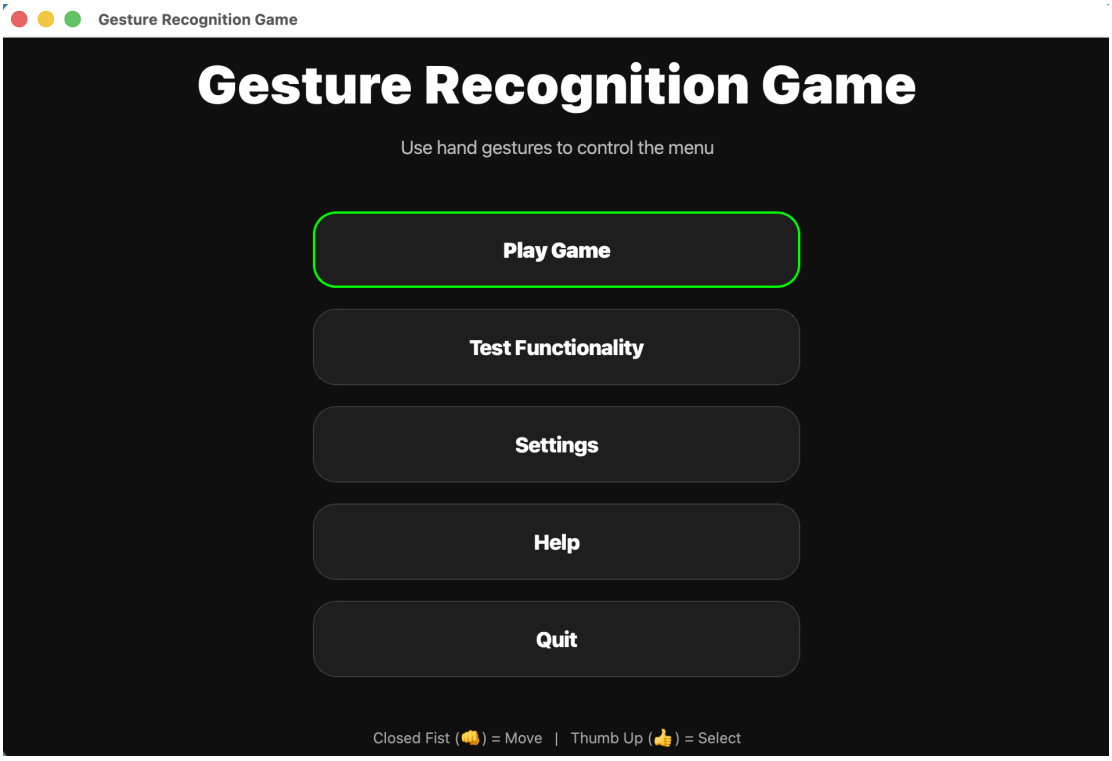


Figure 1: Main menu of the application

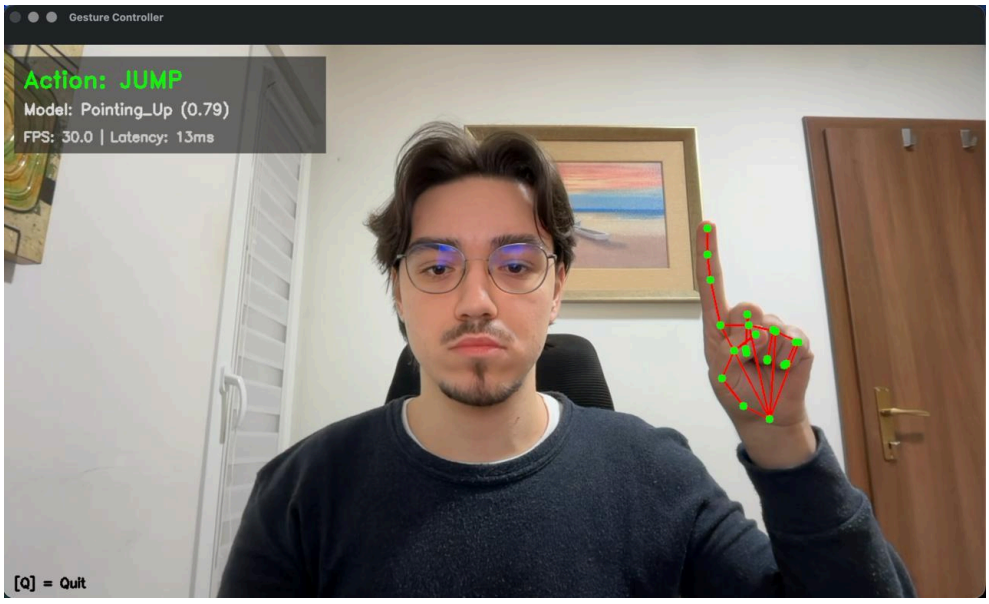


Figure 2: Test mode overlay showing recognized action, FPS, and latency

2. Goals and Objectives

The project objectives are:

- Build a working desktop demo that recognizes hand gestures in real time.
- Integrate the recognizer with a menu-driven GUI that can be operated without mouse/keyboard.
- Provide a testing mode that displays recognition output and performance metrics.
- Support multiple recognition strategies (pre-trained model and hybrid logic).
- Translate recognized gestures into high-level actions and, optionally, into keyboard events for external game control.
- Achieve stable interaction by applying temporal filtering, debouncing, and cooldown logic.

A secondary objective is to demonstrate clean system design for computer vision applications. The project uses modular components, shared interfaces, and a clear separation between perception (recognition), decision (controller), and presentation (UI).

3. Requirements and Use Cases

3.1 Functional Requirements

- Capture live video frames from a webcam continuously.
- Detect a hand and extract features (landmarks) suitable for recognition.
- Classify gestures into predefined labels.
- Map gesture labels to application actions such as menu navigation and selection.
- Allow switching between recognizer modes in a settings screen.
- Provide a test mode for validation and debugging (gesture labels, FPS, latency).
- Support game profiles and emulate keyboard input based on recognized actions.

3.2 Non-Functional Requirements

- Real-time responsiveness: UI must remain smooth while recognition runs.
- Low latency: actions should feel immediate to the user.
- Robustness: short misclassifications should not trigger actions.
- Maintainability: modular structure and clear interfaces between components.
- Extensibility: easy to add new gestures and new game profiles.
- Portability: ability to package the application into an executable/bundle.

3.3 Use Cases

Use Case 1 (Menu Navigation): The user starts the application and uses gestures to move focus between menu buttons. A confirm gesture selects a button and transitions to the next screen.

Use Case 2 (Model Selection): The user opens Settings and switches between recognition modes (MediaPipe Tasks vs. Hybrid Pose-Based). The selection affects the active recognizer used by the worker.

Use Case 3 (Testing and Calibration): The user opens Test Functionality to observe gesture output, adjust camera placement and lighting, and verify stable recognition before controlling the menu or a game.

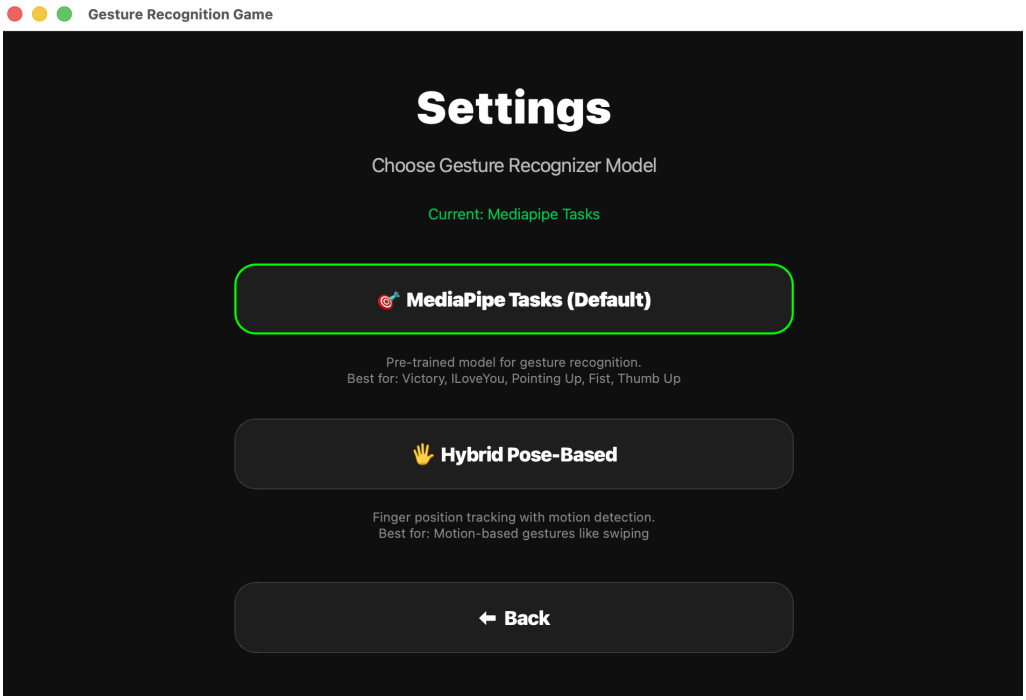


Figure 3: Settings screen showing recognizer selection

4. System Overview

The system converts webcam video into interaction events using a layered pipeline. Recognition runs continuously, but actions are triggered as discrete events only when they are stable and allowed by the controller logic.

Main layers:

- Input Layer: captures frames and prepares them for recognition (format conversion, optional mirroring).
- Recognition Layer: produces gesture labels (and optionally confidence values and landmarks).
- Controller Layer: converts labels into actions and enforces stability (debouncing/cooldowns).

- UI/Output Layer: updates GUI focus/selection and emits keyboard events for game control.

5. Technologies and Tools

The project is implemented using the following technologies:

- Python: main implementation language and runtime environment.
- OpenCV: webcam capture and basic frame operations.
- MediaPipe Tasks Gesture Recognizer: pre-trained hand landmark and gesture classification pipeline.
- PyQt: desktop GUI and application flow.
- NumPy: numerical computations for motion and landmark-based rules.
- pynput: keyboard event emulation for game control.
- PyInstaller: packaging the project as a standalone executable/bundle.

6. Background and Theory

6.1 Digital Image Processing Concepts Used

Even though the project uses a pre-trained model for gesture recognition, it still applies important image processing principles. Video is processed frame-by-frame, which is the standard approach for real-time vision. The pipeline may include color conversion, resizing, and mirroring to adapt the camera feed to recognition needs.

The recognition output is a time series (one output per frame). Treating recognition as a temporal signal introduces the need for filtering techniques such as smoothing, majority voting, and debouncing. These are conceptually similar to filtering noisy sensor data.

6.2 Feature Extraction: Landmarks vs. Pixels

A major idea in gesture recognition is choosing the right representation. Direct pixel-based classification requires large models and can be sensitive to background and lighting. Landmark-based representation reduces the problem to geometric structure: each hand is described by keypoints at important joints and fingertips.

With landmarks, gesture recognition can rely on distances and angles between keypoints. This reduces dimensionality and often improves robustness. Landmarks are also interpretable, which makes them suitable for rule-based (hybrid) logic.

6.3 Real-Time Constraints and Human Factors

Real-time systems must minimize latency while maintaining stability. Users perceive delays above a small threshold as unresponsive behavior. At the same time, users will not

tolerate unpredictable repeated actions. Therefore, controller design (cooldowns, confirmation windows) is as important as recognition accuracy.

From a human-computer interaction perspective, limiting the gesture set to a small number of distinct gestures improves learnability and reduces ambiguity. The chosen gestures for selection and navigation are intentionally simple and widely separable.

7. Architecture and Module Design

7.1 Modular Architecture

The system is separated into modules with clear responsibilities: recognition modules, a controller module, UI modules, and an output module. This separation reduces coupling and makes the project easier to debug and extend.

In particular, the recognizer interface allows switching between MediaPipe-based recognition and hybrid recognition without changing the rest of the application. The UI receives actions rather than raw recognition outputs, which keeps UI behavior consistent.

7.2 Components and Responsibilities

Key components include:

- Recognizer interface: common contract for different recognizers.
- MediaPipe recognizer: loads the .task model and performs inference per frame.
- Hybrid recognizer: interprets landmarks and motion to detect custom gestures.
- Recognizer factory: selects the active recognizer based on settings.
- Gesture worker: captures frames and runs inference in a background thread.
- Controller: debounces and rate-limits actions; maps gestures to actions.
- Performance monitor: measures FPS and latency.
- Game controller: maps actions to key presses based on the chosen game profile.

```python
class GestureRecognizerInterface(ABC):
    """…

    @abstractmethod
    def process(self, frame_bgr: np.ndarray) -> GestureResult:
        """…
        pass

    @abstractmethod
    def draw_landmarks(self, frame: np.ndarray, landmarks: Any) -> np.ndarray:
        """…
        pass

    @property
    @abstractmethod
    def name(self) -> str:
        """Return the name of this recognizer implementation."""
        pass

    @property
    @abstractmethod
    def keyMap(self) -> dict[str,str]:
        """Return the gesture key map of this recognizer implementation."""
        pass

    def cleanup(self) -> None:
        """…
        pass
```

Listing 1: Recognizer interface used by both MediaPipe and Hybrid recognizers

```python
class RecognizerFactory:
    """…
    @staticmethod
    def create(
        recognizer_type: RecognizerType=RecognizerType.MEDIAPIPE_TASKS,
        model_path: Optional[str] = None,
        min_score: float = 0.60,
        mirror_view: bool = True,
        **kwargs,
    ) -> GestureRecognizerInterface:
        """…
        if recognizer_type == RecognizerType.MEDIAPIPE_TASKS:
            if model_path is None:
                model_path = asset_path("gesture_recognizer.task")
            return GestureRecognizerMP(
                model_path=model_path,
                min_score=min_score,
                mirror_view=mirror_view,
                **kwargs,
            )

        elif recognizer_type == RecognizerType.HYBRID_POSE:
            return HybridGestureRecognizer(
                min_detection_confidence=min_score,
                mirror_view=mirror_view,
                **kwargs,
            )

        else:
            raise ValueError(f"Unknown recognizer type: {recognizer_type}")
```

Listing 2: RecognizerFactory selecting the active recognizer based on settings

7.3 Data Flow Between Modules

A typical data flow is: Frame → Recognizer → Gesture label (and confidence) → Controller → Action → UI update / Keyboard output. This pipeline is designed so that each stage is testable independently. For example, the controller can be unit-tested using synthetic gesture sequences.

8. Gesture Recognition Pipeline

8.1 Frame Acquisition and Preprocessing

Frames are captured continuously from the webcam. For user friendliness, a mirrored view can be enabled so that left/right movements match the user's perspective. Frames are converted from BGR (OpenCV default) to RGB depending on the requirements of the recognizer implementation.

To balance accuracy and speed, camera resolution is tuned. Higher resolution did improve landmark precision but increased processing cost. A practical approach is to select a moderate resolution that maintains interactive FPS while keeping the hand sufficiently detailed.

8.2 Recognition Step

The recognition step takes each frame and produces a gesture label. In MediaPipe mode, this includes internal hand detection and landmark estimation. In hybrid mode, landmarks are used explicitly by rule logic. The raw prediction per frame is not yet suitable for direct control because it can flicker.

8.3 Temporal Filtering and Confirmation

We user Temporal filtering to reduce flicker by aggregating multiple frames. Common strategies include requiring N consecutive frames of the same label, applying confidence thresholds, or using majority voting over a short history buffer. The system also uses edge-trigger logic for actions like SELECT, where triggering should occur only once per gesture event.

Confirmation windows are especially useful for 'select' actions. For example, the system requires a thumbs-up gesture to be stable for a short duration before confirming. This reduces accidental selections from brief misclassifications.

8.4 Action Emission and Scheduling

After stabilization, gesture labels are mapped to a fixed set of actions. Actions are scheduled through the controller so that they occur at a controlled pace. For example, navigation actions may repeat with a small delay, while selection actions trigger once.

Listing 3: Worker loop: webcam capture → recognizer inference → filtering → UI update signals

```python
class UIGestureWorker(QThread):
    def run(self):
        cap = cv2.VideoCapture(0)
        cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
        cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

        if not cap.isOpened():
            return

        try:
            while self.running:
                ret, frame = cap.read()
                if not ret:
                    time.sleep(0.01)
                    continue

                result = self.recognizer.process(frame)
                action = result.action

                # Emit only on IDLE -> ACTION (edge trigger)
                now = time.time()
                if action != "IDLE" and self.last_action == "IDLE":
                    if (now - self.last_emit_time) >= self.cooldown_s:
                        self.action_signal.emit(action)
                        self.last_emit_time = now

                self.last_action = action
                time.sleep(0.005)

        finally:
            cap.release()
            # Clean up the dedicated recognizer
            if self.recognizer:
                self.recognizer.cleanup()
```

9. MediaPipe Tasks Gesture Recognizer

9.1 Conceptual Model

MediaPipe's gesture recognizer is a pre-trained pipeline optimized for real-time inference. Conceptually, it performs: (1) hand detection, (2) hand landmark estimation with 21 keypoints, and (3) gesture classification based on landmark features. This approach is efficient because classification operates on compact geometric features rather than full image pixels.

9.2 Landmark Representation (21 Keypoints)

The 21 landmarks represent the wrist plus finger joints and fingertips. With these keypoints, the system can compute features such as finger extension patterns, relative fingertip distances, and overall hand orientation. Stable landmark tracking across frames is critical for consistent recognition.

Landmark-based representations typically generalize better than raw-pixel classifiers in environments with changing backgrounds. However, landmarks can degrade under poor lighting or partial occlusion. Test mode helps users identify conditions where landmarks are reliable.

9.3 Integration into the Application

The application loads the pre-trained model file (gesture_recognizer.task) from the assets directory and initializes the recognizer. For each frame, it runs inference and retrieves the current gesture label. That label is forwarded to the controller, which applies stability logic and produces actions.

Listing 4: MediaPipe recognizer initialization and per-frame inference call

```python
class GestureRecognizerMP(GestureRecognizerInterface):
    def process(self, frame_bgr: np.ndarray) -> GestureResult:
        """
        Process a BGR frame and return gesture result.
        """
        if self.mirror_view:
            frame_bgr = cv2.flip(frame_bgr, 1)

        # Convert to RGB for MediaPipe
        rgb = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2RGB)
        mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=rgb)

        # Generate unique timestamp
        timestamp_ms = int(time.time() * 1000)
        if timestamp_ms <= self._last_timestamp_ms:
            timestamp_ms = self._last_timestamp_ms + 1
        self._last_timestamp_ms = timestamp_ms

        # Run async recognition
        self._recognizer.recognize_async(mp_image, timestamp_ms)

        # Get results
        label, score = self._get_top_label()
        action = self._label_to_action(label, score)
        landmarks = self._get_hand_landmarks()

        return GestureResult(
            frame=frame_bgr,
            action=action,
            raw_label=label,
            confidence=score,
            landmarks=landmarks,
        )
```

9.4 Model Configuration Parameters

In a typical configuration, the recognizer exposes parameters that influence behavior, such as maximum number of hands, detection confidence thresholds, and running mode.

These parameters can affect both accuracy and performance. For example, tracking multiple hands increases compute cost but enables more advanced interactions.

The project's settings screen provides a user-facing mechanism to choose the recognizer mode. Additional parameters can be exposed in the future if more control is desired.

```python
class GestureRecognizerMP(GestureRecognizerInterface):
    """
    Gesture recognizer using MediaPipe Tasks API.

    Uses a pre-trained .task model file for accurate gesture recognition.
    Good for recognizing specific hand poses with high confidence.
    """

    def __init__(
        self,
        model_path: Optional[str] = None,
        min_score: float = 0.60,
        max_hands: int = 1,
        mirror_view: bool = True,
    ):
        """
        Initialize the MediaPipe Tasks recognizer.

        Args:
            model_path: Path to gesture_recognizer.task file (uses default if None)
            min_score: Minimum confidence threshold (0.0 - 1.0)
            max_hands: Maximum number of hands to detect
            mirror_view: Whether to flip the frame horizontally
        """
        if model_path is None:
            model_path = asset_path("gesture_recognizer.task")
```

10. Hybrid Pose/Motion Recognizer

10.1 Motivation

While pre-trained gesture classification works well for standard gestures, interactive systems often need motion gestures such as swipes, or custom gestures not present in the model. The hybrid recognizer complements MediaPipe by combining pose rules with motion tracking over time.

10.2 Pose-Based Rules

Pose rules use geometric relationships between landmarks. A common strategy that we use is to check whether fingertips are above/below intermediate joints in a normalized coordinate system to infer whether a finger is extended. Distances between fingertips and the palm center can also be used to distinguish open vs closed hand states. But we decided to stick to relying on the distance between the fingertips and the PIP or intermediate joints since it allowed for more flexible gesture recognition.

Rule-based logic is interpretable: when a classification is wrong, we tuned and adjusted the hyperparameters such as the hand_detection_confidence the tracking confidence and the movement threshold for sensitivity tuning.

```python
class HybridGestureRecognizer(GestureRecognizerInterface):
    """
    Gesture recognizer using pose-based detection.

    Uses MediaPipe Hands to track finger positions and classify
    gestures based on which fingers are extended and their movement.
    More stable for motion-based gestures like swiping left/right.
    """

    def __init__(
        self,
        min_detection_confidence: float = 0.5,
        min_tracking_confidence: float = 0.5,
        movement_threshold: float = 0.05,
        mirror_view: bool = True,
        debug: bool = False,
    ):
        """
        Initialize the pose-based recognizer.

        Args:
            min_detection_confidence: MediaPipe hand detection confidence
            min_tracking_confidence: MediaPipe hand tracking confidence
            movement_threshold: Minimum movement delta to trigger LEFT/RIGHT
            mirror_view: Whether to flip the frame horizontally
            debug: Print debug information
        """
```

10.3 Motion Detection

Motion-based rules in this implementation track the index fingertip (landmark 8) across consecutive frames. The system computes a displacement delta between the current and previous frame positions, comparing the horizontal movement (delta_x) against a configurable threshold.

If the horizontal displacement exceeds the threshold (default 0.05 = 5% of normalized frame width), the system classifies the motion as a LEFT or RIGHT swipe. The implementation also checks that horizontal movement dominates vertical movement (abs(delta_x) > abs(delta_y)) to prevent diagonal motions from triggering unintended swipes.

The default threshold of 0.05 was chosen to balance sensitivity and stability:

Too low → small hand tremors or tracking jitter cause accidental triggers

Too high → requires exaggerated, uncomfortable swipe motions

Tracking State Management

The system maintains state between frames:

When no hand is detected for multiple frames, tracking state resets to None

On the first frame after reset, the system initializes tracking without emitting a gesture

This frame-to-frame delta approach (rather than a longer temporal window) provides low-latency response, making swipe gestures feel immediate. The tradeoff is that gestures must be performed with a single decisive motion rather than accumulated over time.

Listing 5: Hybrid motion detection: tracking landmark history and computing displacement

```python
class HybridGestureRecognizer(GestureRecognizerInterface):
    def process(self, frame_bgr: np.ndarray) -> GestureResult:

        if self.mirror_view:
            frame_bgr = cv2.flip(frame_bgr, 1)

        rgb = cv2.cvtColor(frame_bgr, cv2.COLOR_BGR2RGB)
        results = self._hands.process(rgb)

        # No hand detected
        if not results.multi_hand_landmarks:
            self._frames_without_hand += 1

            if self._frames_without_hand == 30 and self.debug:
                print("  ⚠ No hand detected - show your hand to the camera")

            # Reset tracking
            self._prev_index_x = None
            self._prev_index_y = None

            return GestureResult(
                frame=frame_bgr,
                action="IDLE",
                raw_label=None,
                confidence=0.0,
                landmarks=None,
            )

        # Hand detected
        if self._frames_without_hand >= 30 and self.debug:
            print("  ✓ Hand detected")
        self._frames_without_hand = 0

        hand_landmarks = results.multi_hand_landmarks[0]

        # Classify pose
        gesture, confidence = self._classify_pose(hand_landmarks)

        return GestureResult(
            frame=frame_bgr,
            action=gesture,
            raw_label=gesture,  # Same as action for pose-based
            confidence=confidence,
            landmarks=hand_landmarks.landmark,  # Return list of landmarks, not NormalizedLandmarkList
        )
```

11. Gesture-to-Action Mapping and Controller Logic

11.1 Gesture Labels vs. Actions

The system separates recognizer outputs (gesture labels) from application intent (actions). This improves maintainability: if a recognizer changes its label set, the mapping layer can be updated without changing the rest of the application.

11.2 Debouncing and Cooldowns

Real-time recognition produces repeated outputs while a gesture is held. If the application reacted to every frame, it would trigger actions continuously and become unusable. The controller solves this using cooldowns (time-based blocking), debouncing (trigger on transitions or after confirmation), and multi-frame stability checks.

Cooldown timing must be chosen carefully. A short cooldown makes navigation responsive but can still feel too fast; a long cooldown improves stability but may feel sluggish. The project uses cooldown values that provide a smooth balance for menu navigation and game control.

11.3 Menu Navigation Behavior

Menu navigation is implemented using a focus model. One menu item is focused at a time. Navigation actions move the focus up/down, and the SELECT action activates the focused item. This interaction style matches typical controller-based interfaces and is well suited to gesture input.

Listing 6: Controller logic: debouncing, cooldown timers, and action emission

```
class GameController:
    def __init__(self):
        self.keyboard = KeyController()
        self.cooldown = 0.12
        self.last_press_time = 0.0

    def execute_action(self, action, profile):
        keymap = self._get_profile_keymap(profile)
        if action not in keymap:
            return

        now = time.time()
        if now - self.last_press_time < self.cooldown:
            return

        key = keymap[action]
        try:
            self.keyboard.press(key)
            time.sleep(0.05)
            self.keyboard.release(key)
            self.last_press_time = now
        except Exception as e:
            print(f"[Controller] Error pressing key: {e}")
```

12. Game Control Output Layer (Keyboard Emulation)

12.1 Why Keyboard Emulation

To control external games, the system generates keyboard inputs that the games already support. This avoids requiring game-specific APIs and enables portability: any game that responds to keyboard controls can be controlled by mapping actions to keys.

12.2 Key Press Simulation

The keyboard emulation layer sends key-down and key-up events with a short hold duration. This is important because some games may ignore extremely short events. A typical approach is: press key → sleep briefly → release key.

Because gesture recognition is continuous, a cooldown is applied to prevent repeated rapid key presses when the same action is produced across consecutive frames.

12.3 Game Profiles and Keymaps

Different games use different control schemes. The project supports profiles such as Subway Surfers (arrow keys) and Temple Run (WASD). Each profile maps actions (LEFT/RIGHT/JUMP/DUCK/SPACE) to appropriate keys. Adding a new game profile requires only adding a new dictionary mapping.

Listing 7: GameController per-game keymaps (Extenstiable)

```
class GameController:
    def _get_profile_keymap(self, profile):
        if profile == "Subway Surfers":
            return {
                "LEFT": Key.left,
                "RIGHT": Key.right,
                "JUMP": Key.up,
                "DUCK": Key.down,
                "SPACE": Key.space,
            }
        elif profile == "Temple Run":
            return {
                "LEFT": "a",
                "RIGHT": "d",
                "JUMP": "w",
                "DUCK": "s",
                "SPACE": Key.space,
            }
        else:
            return {}
```

13. User Interface and Interaction Design

13.1 UI Design Goals

The user interface is designed to be clear, readable, and gesture-friendly. Because users do not receive tactile feedback, visual feedback becomes essential. The UI highlights the currently selected option and provides guidance about supported gestures.

13.2 Screens and Application Flow

The application includes a main menu with actions such as Play, Test Functionality, Settings, Help, and Quit. From Settings, the user can select the recognition mode. From Play, the user can choose a game profile. Test Functionality displays internal recognition status and performance metrics.

Screen transitions are triggered by actions from the controller. For example, SELECT on the Play button moves to the game selection screen. This makes the application fully operable using gestures.

13.3 Feedback and Learnability

To help users learn the system quickly, the interface displays hints about the gesture mapping, and test mode provides immediate feedback about what the system recognizes. This reduces user uncertainty and makes it easier to adjust hand position and movement style.

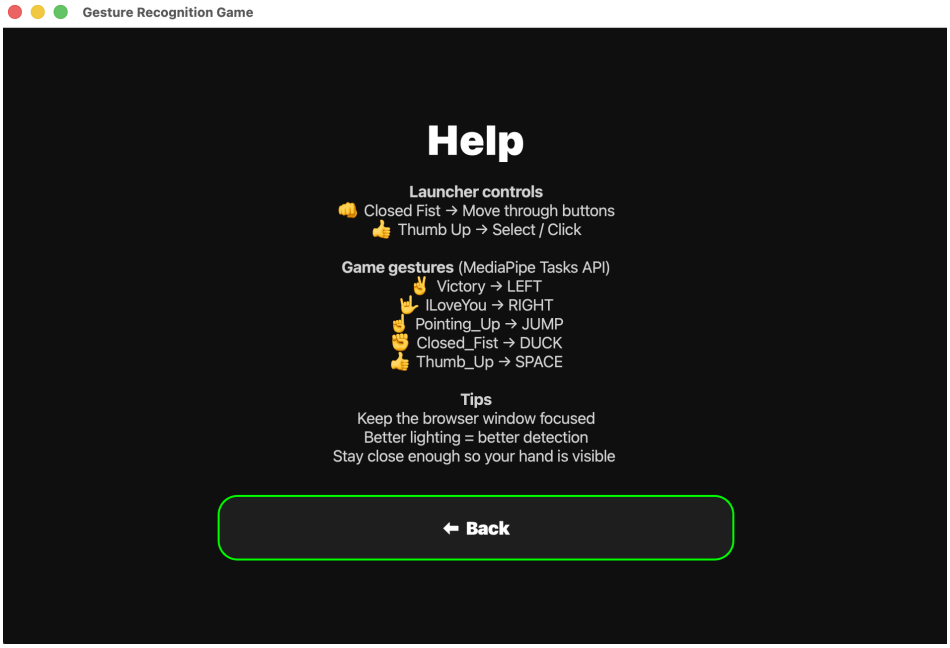


Figure 7: Help menu with tips and control hints

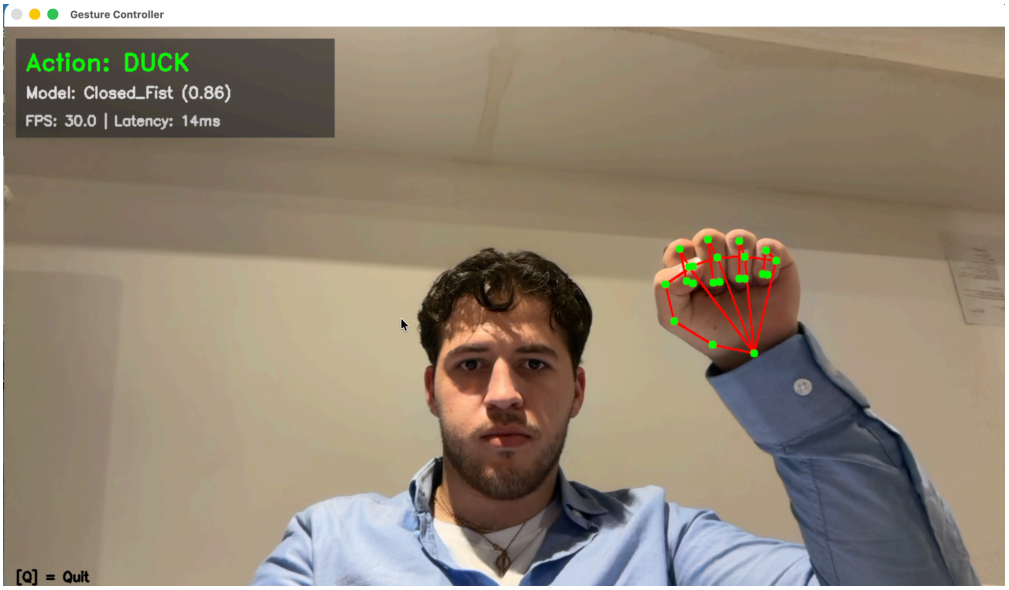


Figure 8: Test mode screen showing action label and performance metrics

14. Implementation Details

14.1 Threading Model and UI Responsiveness

Desktop GUI frameworks use an event loop to handle drawing and user events. If heavy computation runs on the UI thread, the interface becomes unresponsive. To avoid this, the project runs the recognition loop inside a worker thread. The worker captures frames, runs recognition, applies filtering, and emits results to the UI.

The UI receives results as events/signals and updates visual state accordingly. This design keeps the interface smooth even when recognition workload fluctuates.

Listing 8: Worker thread setup and UI signal emission (INSERT CODE HERE)

Paste a relevant code excerpt here (15–60 lines).

14.2 Path Handling for Assets

Computer vision projects often rely on external model files. The application must locate the model file correctly both in development runs and in packaged builds. The project uses a dedicated path utility to resolve the correct model path, preventing 'file not found' issues after packaging.

Listing 9: Asset path resolution for gesture_recognizer.task

```python
class GestureRecognizerMP(GestureRecognizerInterface):
    """…

    def __init__(
        self,
        model_path: Optional[str] = None,
        min_score: float = 0.60,
        max_hands: int = 1,
        mirror_view: bool = True,
    ):
        """…
        if model_path is None:
            model_path = asset_path("gesture_recognizer.task")
```

```python
def asset_path(filename: str) -> str:
    # When bundled by PyInstaller
    if hasattr(sys, "_MEIPASS"):
        base = sys._MEIPASS
        return os.path.join(base, "app", "assets", filename)

    # Normal run from source
    here = os.path.dirname(os.path.abspath(__file__))  # app/core
    return os.path.join(here, "..", "assets", filename)
```

14.3 Error Handling Strategy

In a real-time application, failures such as missing camera devices, missing model files, or initialization errors should not crash the program silently. The project includes error handling around key steps such as model loading and key press emission. Errors are printed or surfaced so debugging is possible.

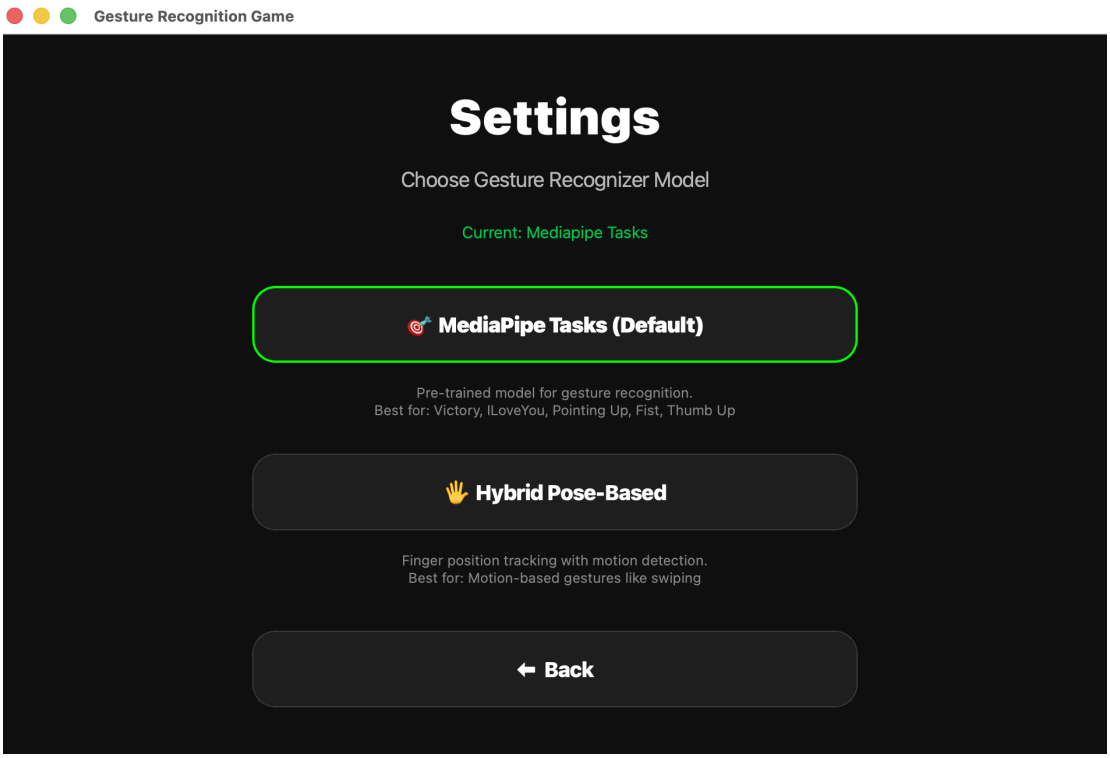
Common failure cases include webcam permissions on the operating system and asset bundling issues in packaged builds. These are addressed through deployment notes and hooks.

```
from PyInstaller.utils.hooks import (
    collect_submodules,
    collect_data_files,
    collect_dynamic_libs,
)

# Collect ALL mediapipe modules (including mediapipe.tasks.c)
hiddenimports = collect_submodules("mediapipe")

# Collect models/config files inside mediapipe
datas = collect_data_files("mediapipe", include_py_files=True)

# Collect native libs (.so/.dylib/.dll) that mediapipe needs
binaries = collect_dynamic_libs("mediapipe")
```

15. Configuration and Profiles

15.1 Recognizer Mode Selection

The Settings screen allows the user to switch between recognizer modes. This choice determines which recognizer implementation the factory instantiates. Switching modes enables comparing stability and performance trade-offs between the pre-trained model and hybrid logic.



15.2 Gesture Set and Action Set

The project uses a set of gestures that are mapped to a stable set of actions. Keeping the action set small improves reliability and reduces ambiguity. Actions such as LEFT/RIGHT/JUMP/DUCK/SPACE provide a clear interface for both menu navigation and game control.

15.3 Game Profiles

Game profiles define how actions map to keyboard keys for a specific game. This design supports extensibility: new profiles can be added without touching the recognition pipeline. Profiles also allow using different control schemes for different games.

16. Performance, Evaluation, and Results

16.1 Performance Metrics

The project measures frames per second (FPS) and latency (time per inference). FPS indicates processing throughput, while latency indicates how quickly the system can react to a user gesture.

16.2 Evaluation Methodology

Evaluation is performed in test mode under different conditions: varying lighting, varying hand distance, different backgrounds, and different gesture speeds. Both recognizer modes are tested to compare stability and responsiveness.

16.3 Observations

The system performs best when the hand is clearly visible, lighting is adequate, and the camera view is stable. MediaPipe generally provides strong landmark tracking and stable classification for standard gestures. Hybrid recognition can add flexibility for motion gestures, but it depends on stable landmark detection and properly chosen thresholds.

Stability mechanisms (cooldown and debouncing) are critical. They turn continuous recognition output into discrete events, making both UI navigation and game control predictable.

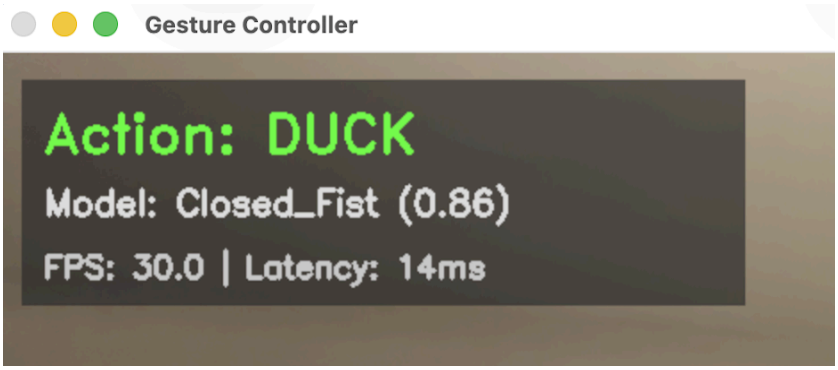


Figure 10: Screenshot showing FPS/latency overlay during evaluation

17. Packaging and Deployment

17.1 Packaging Challenges

Packaging computer vision applications can be challenging because model files and internal resources may not be included automatically. A common issue is that the application works locally but fails after packaging due to missing assets or incorrect paths.

17.2 PyInstaller Strategy

The project uses PyInstaller to create a standalone build. A custom hook is included to ensure that MediaPipe dependencies and resources are bundled correctly. Asset path handling ensures the model file is found in both development and packaged environments.

17.3 Deployment Notes

For successful deployment, webcam permissions must be enabled on the target system. When using keyboard emulation, the target game window must be focused so that key events are received by the game.

18. Limitations, Risks, and Mitigations

18.1 Limitations

- Recognition quality depends on lighting, camera quality, and background complexity.
- Occlusions and extreme hand rotations may reduce landmark accuracy.
- Fast motion can introduce blur and classification flicker.
- Keyboard emulation requires the correct target window to be focused.
- Gesture vocabulary is constrained by available model labels and stability.

18.2 Risks

Because the system can send global keyboard events, accidental triggers are possible if the wrong window is focused. Short misclassifications may also trigger unintended actions if stability rules are too permissive.

18.3 Mitigations

Cooldown windows and debouncing reduce the risk of repeated or accidental actions. The UI provides feedback about the current mode and action, and test mode can be used to verify recognition stability before enabling game control.

19. User Guide (How to Run and Use)

19.1 Installation

To run the project in development mode, install dependencies using the provided requirements file. Ensure that the operating system grants camera permissions to Python. If a virtual environment is used, activate it before running the application.

19.2 Running the Application

Start the application using the main launcher script. After startup, the main menu is displayed. Use gestures to move between menu options and select items. If recognition appears unstable, open Test Functionality to observe current outputs and adjust lighting/distance.

19.3 Using Settings and Profiles

Open Settings to choose the recognizer mode. Open Play to select a game profile. When game control is enabled, bring the game window into focus and perform gestures to send corresponding key presses.