MultiModalMan: An Interactive Hangman Game

Andrea Tarricone
tarricone.1888181@studenti.uniroma1.it
University "La Sapienza"
Rome, Lazio, Italy

Lucian Dorin Crainic
crainic.1938430@studenti.uniroma1.it
University "La Sapienza"
Rome, Lazio, Italy

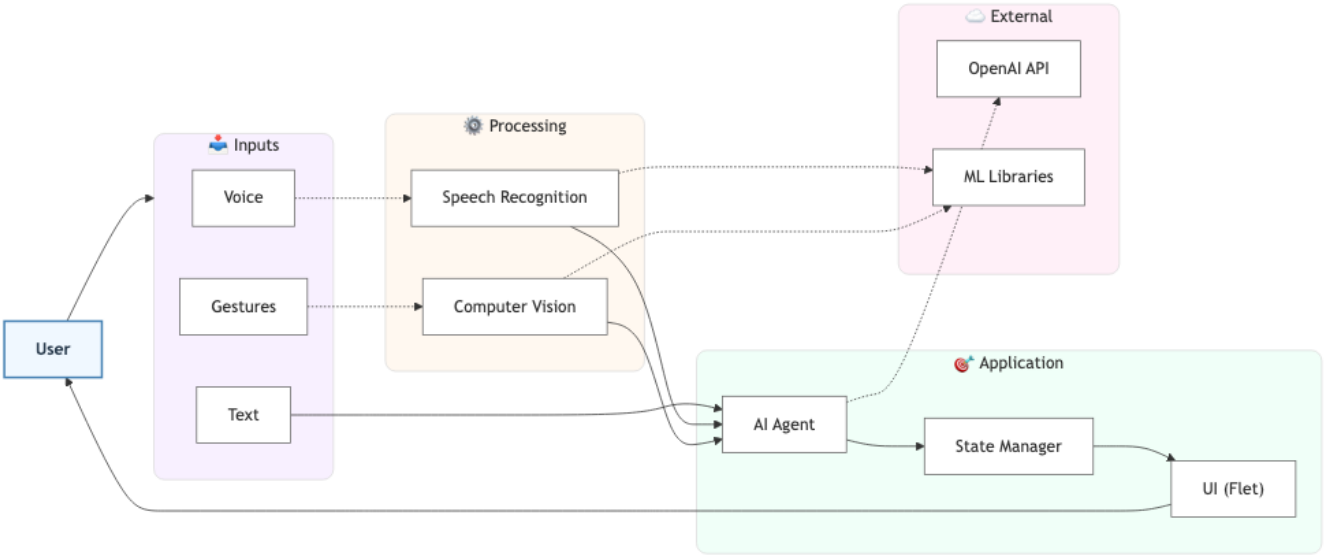

Figure 1. High level overview of the MultiModalMan game system showing the flow from user inputs through processing layers to the core application.

Abstract

MultiModalMan is a reimagining of the classic *hangman game* that lets players guess letters and control the interface by speaking or using hand gestures as well as by typing. The system is built in *Python* around three modules: a state manager that synchronises all inputs, an **AI agent** for *natural language processing*, and a *Computer Vision pipeline* for *real time gesture recognition*. Users can switch freely between modalities during play, making the game accessible and engaging for a wide audience. The application is implemented using the *Flet* framework, which provides a responsive web interface that runs in any modern browser. This paper describes the architecture, design patterns, and implementation details of the system, highlighting its innovative use of multimodal interaction to enhance user experience.

1 Introduction

Digital games have become a mainstream medium for both entertainment and learning, yet they remain unevenly playable. Industry reports estimate that *20–30 %* of the global player base lives with at least one disability, and almost half of those players already engage with games despite a range of access barriers [1]. Ensuring that games are accessible is therefore both an ethical imperative and a clear educational and commercial opportunity.

Research in multimodal human computer interaction (HCI) shows that combining complementary input channels speech, gesture, gaze, or haptics lowers access barriers and improves task performance. A recent survey notes consistent reductions in error rate, task time, and cognitive load when users can blend modalities rather than rely on a single channel [2]. Game specific studies echo these findings: gesture based controllers enable players with motor impairments to execute directional actions that would be impossible on a traditional keyboard [5], and experiments that fuse voice commands with hand gestures yield the highest efficiency and perceived naturalness among the tested conditions [3]. Meanwhile, advances in AI speech processing continue to broaden accessibility by adapting to diverse accents and speech differences [4].

MultiModalMan responds directly to this research agenda. The project reimagines the classic Hangman game as a language learning tool that can be played interchangeably through:

- spoken commands processed by an AI speech pipeline;
- hand-gesture input captured via a webcam and MediaPipe;
- conventional keyboard typing.

An embedded AI agent maintains the dialogue flow and adapts hints in real time, while a finite state machine keeps all three modalities synchronised so players can switch fluidly between them without losing context. The project pursues the following goals:

1. demonstrate how a lightweight multimodal pipeline can be integrated into a small educational game;
2. provide meaningful accessibility options for users with motor or speech restrictions;

2 Aim and Scope

MultiModalMan pursues a twofold mission: to illustrate how multimodal interfaces lower accessibility barriers and to turn that insight into an engaging, language learning game. Traditional mouse and keyboard designs routinely sideline entire user groups. Blind players cannot perceive graphical feedback; Deaf players can read on screen text but gain little from audio narration; users with motor impairments may struggle with precise key presses. By blending speech, gesture and text input, our project offers redundant and therefore more inclusive paths through the game.

Accessibility Objectives

Multiple channels Voice, hand gestures and on screen buttons are interchangeable so that no single sense or motor skill becomes a blocker.

Low reading load Visual icons and spoken prompts replace verbose text wherever possible.

Complementarity When one modality fails (e.g. speech in a noisy room) another can take over without interrupting play.

Technical Targets

Accurate input Real time recognition of spoken letters and hand drawn glyphs.

Context-aware logic An AI Agent that interprets free form utterances and drives adaptive hints.

Unified control A central State Manager that keeps the three modalities synchronised and responsive.

3 Requirements

The following tables specify the essential capabilities (*Functional*) and quality targets (*Non Functional*) that **MultiModalMan** must meet to deliver an inclusive, responsive and maintainable experience.

Table 1. Functional requirements

ID	Description
F1	The user can start a new game at any time.
F2	The system selects a random word or User suggests a word to the AI agent.
F3	Letters can be guessed through voice commands, hand gestures or keyboard input.
F4	The interface shows the current word state and the number of wrong attempts.
F5	The system announces the outcome (win or loss) when the game ends.
F6	A new round can be launched without restarting the application.

Table 2. Non functional requirements

ID	Description
NF1	The user interface must be clear and readable for all age groups.
NF2	Gesture recognition covers the full 26 letter English alphabet
NF3	The game logic remains consistent when inputs are invalid or ambiguous.

4 Use Case Model

A use case captures how an external actor perceives the system. In **MultiModalMan** that actor is the *Player*, who may speak, gesture, or type.

Primary scenarios : UC1 Start Game, UC2 Select Input Mode, UC3 Guess Letter, UC4 Display Word State, UC5 Handle End of Game, UC6 Start New Round.

Table 3. Figure 2 displays the UC1 diagram, which illustrates the primary interactions between the Player and the system.

ID	Actor	Goal	Outcome
UC1	Player	Launch the application.	Home screen shown
UC2	Player	Choose voice, gesture, or keyboard.	Selected channel
UC3	Player*	Guess a letter.	Word state updated
UC4	System	Present current word and errors.	Updated status
UC5	System	Detect win or loss.	Outcome announced

* Variants: Voice +STT, Gesture +CNN, or direct keyboard input.

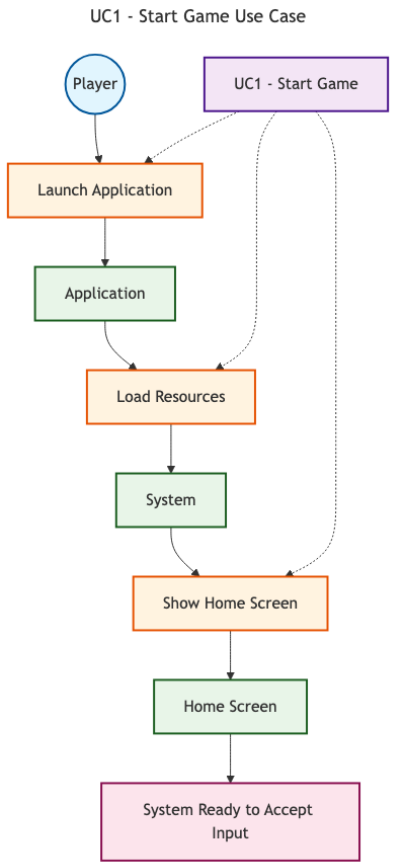

Figure 2. Use case diagram for the **MultiModalMan** game.

5 System Architecture

MultiModalMan is implemented as a web application using Python and the Flet framework. The architecture is designed to run on Windows, Linux, and macOS systems, requiring only a standard webcam and microphone for full multimodal functionality.

High Level Architecture

The system follows a layered architecture pattern with clear separation between the presentation layer (frontend), business logic (controller), and specialized processing services (backend). This design ensures modularity, maintainability, and enables independent development of different components.

The application consists of three main architectural layers:

Frontend Layer Web based user interface built with Flet, handling all user interactions and visual feedback

Controller Layer AI powered game agent that manages game flow and natural language processing

Backend Layer Computer vision and machine learning models for gesture recognition

Module Structure

The project is organized into three primary modules, each serving distinct architectural roles:

`frontend/`. The frontend module implements the web based user interface using the Flet framework. It contains the complete presentation layer and game state management:

`main.py` Application entry point that initializes the web interface and coordinates between components

`src/app/` Core application logic including:

- `game_logic.py`: `HangmanGame` class managing game rules, word tracking, and win/loss conditions
- `state_manager.py`: `GameStateManager` singleton implementing the observer pattern for real time UI updates

`src/components/` Modular UI components:

- `game_panel.py`: Main game display with word visualization and controls
- `media_controls.py`: Multimodal input interface (voice, gesture, chat)
- `hand_drawing_recognition.py`: Computer vision integration for gesture input
- `display.py`, `layout.py`: Responsive UI layout and visual components

`controller/`. The controller module houses the AI powered game agent that serves as an intelligent intermediary between user inputs and game actions:

`agent.py` OpenAI conversational agent that:

- Interprets natural language commands and converts them to game actions
- Maintains conversation context and provides adaptive hints
- Manages game flow through structured function calls
- Integrates with the frontend's `GameStateManager` for state synchronization

The agent uses a singleton pattern to ensure consistency between frontend and controller state management, enabling seamless communication between the AI logic and the User Interface.

`backend/`. The backend module provides specialized machine learning capabilities for gesture recognition:

`models/` Pre trained neural network models:

- `letter_recognition.h5`: CNN model trained on EMNIST dataset for hand drawn letter classification

`src/` Computer vision processing pipeline:

- `hand_model.py`: `HandModel` class for loading and running inference on hand drawn letters
- `tracker.py`: Real time hand tracking and gesture detection using OpenCV

Data Flow Architecture

The system implements a centralized state management pattern where all user inputs regardless of modality flow through the `GameStateManager`. This ensures consistent game state and enables seamless switching between input methods:

1. **Input Processing**: Voice (speech recognition), gestures (CNN inference), or text input are processed by their respective handlers
2. **State Management**: All inputs are channeled through the `GameStateManager`, which validates actions and updates game state
3. **Observer Notification**: UI components are automatically updated via the observer pattern when state changes occur
4. **AI Integration**: The OpenAI agent maintains parallel state synchronization for natural language processing and adaptive responses

This architecture ensures that users can switch fluidly between input modalities without losing game context or experiencing state inconsistencies.

Deployment Architecture

MultiModalMan is deployed as a web application that runs locally and serves a browser-based interface. The deployment model supports rapid development and cross-platform compatibility:

Local Server The Flet framework creates a local web server that hosts the application interface

Browser Client Users interact through any modern web browser, requiring no additional software installation

External Services The application integrates with external APIs:

- OpenAI API for natural language processing and conversational AI
- Google Speech Recognition API for voice input processing

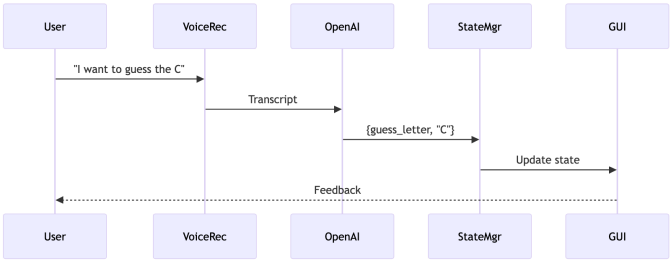

Figure 3. Activity diagram for the voice interaction component, showing how user speech is processed, transcribed, and routed through the state manager to update the game state.

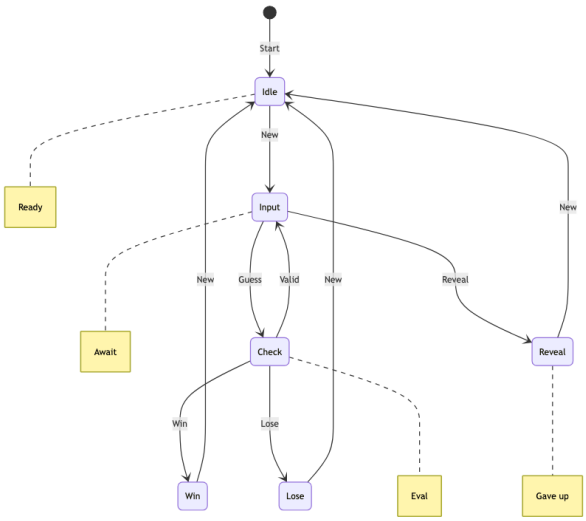

Figure 4. State machine diagram for the `GameStateManager` class, showing the transitions between different game states based on user inputs and actions.

6 Multimodal Interaction

Multimodal interaction is the cornerstone of **MultiModalMan**. Instead of locking players into a keyboard and mouse loop, the game accepts spoken commands, hand drawn letters captured by a webcam, and classic text input. A Python state manager keeps these channels in sync, while an OpenAI-powered agent interprets free-form utterances and adapts feedback to the current game phase.

Voice Interaction

When the user speaks “start game”, “guess C”, or “exit” for instance the audio stream is transcribed in real time (primary engine: `speech_recognition`; fallback: Whisper or the Google Speech API). The recognised text is passed to the state manager, which checks whether the utterance is valid for the current state (setup, guessing, or end of game) and routes it accordingly. Natural phrases such as “how many errors do I have?” are interpreted by the OpenAI agent, which converts them into concrete game actions or informational responses.

Gesture Interaction

A webcam continuously detects the user’s hand. OpenCV isolates the region of interest, normalises it, and forwards the image to a lightweight CNN trained on finger spelled letters. The predicted character is then forwarded to the state manager, which decides whether to accept, reject, or request clarification always providing on screen feedback so users understand what was recognised.

State Manager

The `GameStateManager` class acts as the game's traffic controller. It tracks a finite set of states (initialisation, waiting for input, verifying, end of game) and exposes two public methods `handle_voice_input()` and `handle_gesture()`. Because every modality funnels through this single point, conflicts are resolved centrally: if a gesture and a voice command arrive at the same instant, the manager applies simple time stamping to decide which one to honour first. It also filters commands that make sense only in a given phase (e.g., "new game" is accepted only after a round has finished).

OpenAI Agent

The agent turns raw language into game intents. It maintains short term context current word, remaining attempts, last hint delivered so that a query like "remind me what I've tried" produces a targeted answer. Outputs are JSON objects containing an `action` field (e.g. `"guess_letter"`) and an optional `value`. This structure lets the state manager treat the agent as just another input device, keeping the architecture loosely coupled.

For security, the OpenAI key is injected via the `OPENAI_API_KEY` environment variable (e.g. `export OPENAI_API_KEY=`⟨key⟩ on Unix).

Listing 1. Essential agent scaffold

```
agent = Agent(
    name="hangman_game_agent",
    instructions="High-level rules and prompts (
        truncated for brevity)",
    tools=[
        ("welcome_agent", "Greets and explains
            rules"),
        ("wordsetter_agent", "Chooses the word"),
        ("letter_guesser_agent", "Processes a
            letter guess"),
        ("game_restarter_agent", "Starts a new
            round"),
        ("sync_agent", "Checks active game state")
            ,
    ],
)
```

7 GUI Implementation

Application Architecture

The GUI is built using the Flet framework, creating a responsive web based interface that runs in the browser. The application follows a two panel layout design where the main game display occupies the left panel and the multimodal input controls are positioned on the right panel.

To launch the application, use the following command from the project root:

Listing 2. Starting the Frontend Application

```
flet run frontend/main.py --web
```

This opens the game in the default web browser as a responsive web application.

Main Components

The interface is organized into modular components, each handling specific aspects of the game:

Component	Responsibility
`GamePanel`	Main game interface with word display, hangman visual, and control buttons.
`GameDisplay`	Shows current word state, guessed letters, remaining attempts, and game status.
`HangmanVisual`	Animated hangman drawing that progresses with wrong guesses (7 states).
`MediaControls`	Tabbed interface for voice input, hand drawing, and AI agent chat.
`ManualInput`	Text input field for manual letter guessing with validation.
`AppLayout`	Responsive layout manager handling window resizing and notifications.

Game Panel Layout

The left panel contains the core game interface:

- **Control Buttons**: Three action buttons for starting new games, resetting, and revealing words
- **Word Display**: Shows the target word with revealed letters or underscores for hidden letters
- **Game Information**: Displays guessed letters, remaining attempts, and current game status
- **Hangman Visual**: Animated SVG style drawing showing game progress (0-6 wrong guesses)
- **Manual Input**: Text field for direct letter input with real time validation

Multimodal Input Interface

The right panel provides three input modalities through a tabbed navigation interface:

Agent Chat Natural language conversation with an AI agent that manages the game, processes letter guesses, and provides guidance. Includes voice to text capability for hands free interaction.

Voice Input Direct voice recognition for letter guessing. Users say "Letter X" to guess a specific letter, with visual feedback showing the recognized input and confidence levels.

Hand Drawing Computer vision based letter recognition using webcam input. Users draw letters in the air, which are captured, processed through a CNN model, and converted to game guesses.

State Management and Observer Pattern

The application uses a centralized state management system with an observer pattern for real-time UI updates:

Listing 3. Observer Pattern Implementation

```
class GameStateManager:
    def add_observer(self, callback: Callable[[
        GameState], None]) -> None:
        """Add a callback function to be called
            when game state changes"""
        if callback not in self._observers:
            self._observers.append(callback)

    def _notify_observers(self, state: GameState)
        -> None:
        """Notify all observers of a state change
            """
        for callback in self._observers:
            callback(state)

# Components register as observers
self.state_manager.add_observer(self.
    on_state_changed)
```

This ensures that all UI components automatically update when the game state changes, whether triggered by manual input, voice commands, hand gestures, or AI agent actions.

Component Implementation Details

GameDisplay Component. The `GameDisplay` component handles the visual representation of the current game state:

Listing 4. GameDisplay Update Logic

```
def update(self, state):
    # Handle revealed word display
    if state.game_status == "revealed" and state.
        secret_word:
        self.word.value = " ".join(state.
            secret_word)
        self.word.color = config.COLOR_PALETTE["
            error"]
    else:
        self.word.value = state.display_word
        self.word.color = None
    self._update_status(state)
```

HangmanVisual Component. The hangman visual progresses through seven distinct states, from an empty gallows to a complete figure:

Listing 5. Hangman Visual State Management

```
def update_state(self, wrong_guesses):
    """Update the hangman visual based on number
        of wrong guesses"""
    wrong_guesses = max(0, min(wrong_guesses, 6))
    self.current_state = wrong_guesses
    self.content = self.states[wrong_guesses]
    self.update()
```

MediaControls Navigation. The multimodal interface uses a navigation bar to switch between input methods:

Listing 6. Multimodal Interface Navigation

```
def _handle_tab_change(self, e):
    """Handle navigation rail tab change"""
    selected_index = e.control.selected_index
    if selected_index == 0:  # Chat
        self.view_container.content = self.
            chat_view
    elif selected_index == 1:  # Voice
        self.view_container.content = self.
            voice_view
    elif selected_index == 2:  # Drawing
        self.view_container.content = self.
            drawing_view
    self.view_container.update()
```

8 Conclusion and Future Work

Key Achievements

MultiModalMan reimagines the classic paper hangman as an inclusive digital game: players can now interact by speaking, gesturing, or typing, while an AI agent orchestrates the state machine, understands natural language commands, and adapts hints in real time making the experience accessible to a far wider audience than the original pencil and paper version ever reached.

Future Directions

1. Expand the word set and add text to speech for fully voicedriven, multilingual play.
2. Introduce cooperative and time trial multiplayer modes.
3. Provide an interactive onboarding sequence for first time users.
4. Store user profiles and match history to track learning progress.

References

[1] AbleGamers Foundation. 2024. *How the Gaming Industry Is Adapting to the Needs of Gamers With Disabilities.* https://ablegamers.org/how-the-gaming-industry-is-adapting/ Accessed 18 Jun 2025.

[2] Muhammad Zeeshan Baig and Manolya Kavakli. 2020. Multimodal Systems: Taxonomy, Methods, and Challenges. *arXiv preprint arXiv:2006.03813* (2020). https://doi.org/10.48550/arXiv.2006.03813

[3] Lizhou Cao, Huadong Zhang, Chao Peng, and Jeffrey T. Hansberger. 2023. Real-Time Multimodal Interaction in Virtual Reality: A Case Study With a Large Virtual Interface. *Multimedia Tools and Applications* 82, 16 (2023), 1–22. https://doi.org/10.1007/s11042-023-14381-6

[4] Meredith Ringel Morris. 2019. AI and Accessibility: A Discussion of Ethical Considerations. *Commun. ACM* (2019). https://www.microsoft.com/en-us/research/wp-content/uploads/2019/08/a14a-ethics-CACM-viewpoint-arxiv.pdf Viewpoint column, in press.

[5] Atieh Taheri, Ziv Weissman, and Misha Sra. 2021. Design and Evaluation of a Hands-Free Video Game Controller for Individuals With Motor Impairments. *Frontiers in Computer Science* 3 (2021), 751455. https://doi.org/10.3389/fcomp.2021.751455