H2Othello

By Group 12B

BULLIER Gauthier
GNOM Arthur
MATHEU Flavien
VUILLEMIN Diane

Tutor: Simone AONZO

Engineering Degree Semester 6 Project

Contents

List of Figures

1 Introduction

1.1 Project context

The gaming industry is a rapidly expanding market, with over 40% of the global population playing video games on a weekly basis. Beyond entertainment, this sector is leading technological innovation in fields such as graphics processing, artificial intelligence, and human-computer interaction. Additionally, gaming increasingly serves educational and training purposes, demonstrating its broader societal impact. This semester project explores these facets through the development of a game based on Othello on raspberry pi 4. The project is built around four main components:

1. A user-friendly interface that reflects the game's theme and enhances the player experience.

2. Voice control functionality, allowing players to interact with the game using both traditional input devices and voice commands.

3. An artificial intelligence module capable of competing against human players.

4. A communication protocol enabling two players to engage in multiplayer games from separate computers.

The teacher Massimiliano TODISCO, the tutor Simone AONZO and the management coach Pauline GRESSE played a crucial role in providing guidance, clarifications, and support throughout the project's development. Their expertise and insights contributed to the project, ensuring that the implemented solutions met the required standards and best practices.

The project was undertaken at EURECOM, a leading graduate school and research center located in Sophia Antipolis, France. EURECOM specializes in information and communication technologies (ICT), providing a collaborative and innovative environment for students and researchers.

2 Project Organization

2.1 Task Distribution

The work was distributed across the four core technical pillars defined by the S6 project structure:

- **Pillar 1 – Interface Design:** Responsible for the game interface and user interaction using Pygame. Tasks included improving the graphical layout, animations, and responsiveness.
- **Pillar 2 – Artificial Intelligence:** Focused on designing and integrating an AI player using 3 methods.
- **Pillar 3 – Voice Control:** Implemented a real-time speech recognition module using Vosk to allow players to speak their moves. This included preprocessing, recognition, phonetic correction, and integration with the game logic.
- **Pillar 4 – Network Communication:** Developed the TCP communication protocol enabling two Raspberry Pi devices to synchronize gameplay in real time. This included message formatting, error handling, and connection management.

Each group member was assigned one pillar as primary responsibility, while also contributing to the integration and testing of the full system.

2.2 Main Work Phases

The project was carried out over a full semester, with two to three dedicated sessions per week. It was structured into three main phases:

1. **Design and Planning:** Definition of protocol specifications, system architecture, and task allocation. Early experiments with voice control and network setup were conducted in this phase.
2. **Implementation and Integration:** Development of each module separately. And then integrate each part together
3. **Testing and Refinement:** Debugging, performance tuning, test the network protocol with other group and adaptation to Raspberry Pi constraints.

3 Technical Components

3.1 Game Interface (Pillar 1)

Overview

The game interface was designed to be intuitive and visually engaging. It consists of multiple windows and animations that guide the player through various stages of the game, from menu navigation to gameplay and endgame announcements.

Interface Windows

- **Main Game Board:** Displays the 8×8 Othello grid. The visual theme changes based on the AI difficulty level selected.
- **Pre-menu:** Lets the player choose between local or network play.
- **Menu (Local Menu):** Allows the user to choose to play against a human or an AI.
- **Difficulty Menu (AI mode):** Lets the user choose the difficulty of the AI opponent.
- **Network Menu:** Allows the user to choose to host or join a game, assign human or AI players, and input host port values.
- **Network Info:** Shown while a network connection is being established.
- **Endgame Screen:** Displays a custom image announcing the winner of the game.

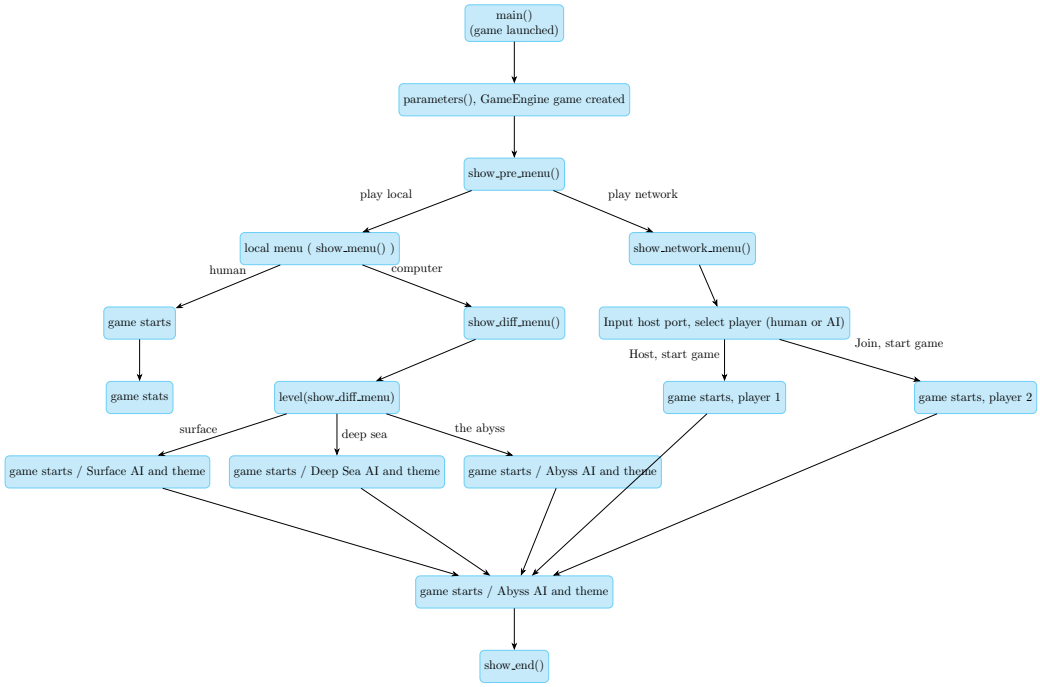
Navigation Flow

Players navigate the interface through a series of interactive windows. Each selection determines the next window displayed.

How windows work

Each window uses images loaded into `pygame` as visual elements. These images are wrapped in `Rect` objects to manage size and position. Rendering happens in the main loop via the `blit()` method. Interactivity is handled by checking if the mouse click position collides with any button `Rect`.

Two primary animations are implemented:



Mapping of the different windows

- **Click Animation:** Buttons visually react to clicks by changing state and color.
- **Background Animations:** Includes alternating background images and movement of fish/shark, updated on each redraw (while loop).

Drawing and Interacting with Pieces

The board is centered and divided into an 8×8 grid. When a player clicks on a cell, the click position is converted to grid coordinates using the board's origin and block size. If the move is legal, the game state is updated and the new piece is drawn in the corresponding grid cell.

Adaptive Window Scaling

The game supports window resizing via `pygame.VIDEORESIZE`. Most objects scale dynamically using custom scaling functions to fit the new window size. However, this is not yet implemented for the main game window as it would have required too much time.

Visual Design

Most visual assets (pieces, boards, backgrounds) were created using a pixel art tool. The goal was to make a coherent design for every part of the game.

3.2 Artificial Intelligence for Gameplay (Pillar 2)

Overview

AI represents a pivotal role in the game. We focused on implementing 3 levels of intelligence for the opponent suiting an inexperienced player seeking fun, and challenging enough for a more advanced player. The three deepnesses of difficulty are named **Surface, Deep Sea** and **The Abyss** respectively for easy, medium and hard mode.
The 3 AIs hold from a super-class `GameAI()` the following methods :

- `performMove()` which computes the best move according to the algorithm used and then call `game.performMove(x,y)` in order to place the piece
- `findBestMove()` a function used by `performMove()` to compute the best move according to the algorithm used

Surface (Greedy AI)

Its goal is to maximize the score at each move. It computes legal moves and chooses the one which flips the maximum of tiles. It is a fast algorithm (**Complexity** : $O(M)$, M the number of legal moves).

Pseudo-code :

Algorithm 1 Greedy AI

```
1: function GreedyBestMove(board, player)
2:     best ← (−∞, −∞), maxFlips ← −∞
3:     for all (x, y) ∈ LegalMoves(board, player) do
4:         flips ← PlacePiece(board, x, y, player , simulate)
5:         if flips > maxFlips then
6:             maxFlips ← flips; best ← (x, y)
7:         end if
8:     end for
9:     return best
10: end function
```

Pro : Easy and fast AI suitable for amateur players.
Con : Very simple to beat.

Deep Sea (MiniMax AI)

For our MiniMax, 3 key mechanisms allowing to be practical for Othello game.

First, the `performMove()` function sets a 3 seconds timer after which the iterative deepening `IDMinimax()` function stops seeking for a best move and return the best move kept in memory. Secondly, Alpha-Beta pruning is implemented with the parameters `alpha` and `beta` : whenever the child value is superior to `beta` and inferior to `alpha`, the remaining siblings are skipped.

The score is attributted to leafs by the `utility()` function featuring 4 evalutaors :

- `pieceDifference()` : the difference between the player and AI's number of pieces on the board
- `mobility()` : If AI has more stable pieces than player (or vice-versa)
- `cornerCaptures()` : If AI has captured more corner than the player (or vice-versa)
- `cornerCloseness()` : If AI has captured more corner closeness than the player (or vice-versa)

Finally, their coefficients are attributed by `phaseWeights`, adjusting the importance of the evaluator according to the phase of the game. Before 30 moves, corner closeness and mobility are the most important criteria for example.

Pseudo-code :

Algorithm 2 Minimax (iterative deepening)

```
1: function BestMove(board, player, limit)
2:     deadline ← now() + limit
3:     depth ← 1
4:     best ← ∅
5:     while now() ≠ deadline do
6:         (v, m) ← AlphaBeta(board, depth, −∞, +∞, player)
7:         if now() = deadline then
8:             best ← m
9:         end if
10:        depth ← depth + 1
11:    end while
12:    return best
13: end function
```

The Abyss (Monte-Carlo tree search)

The Monte-Carlo Tree Search runs entirely inside the function `performMove()`. At the start, `performMove()` sets a timer and clones the current board so the real game state will not change during the search. While time remains, it repeats four internal steps :

- `select` walks down the tree, always choosing the child that maximizes the formula:

$$\mathrm{UCT}(i) = \frac{w_i}{n_i} + C \cdot \sqrt{\frac{\ln N}{n_i}}$$

  where w_i is the number of wins, n_i is the number of visits of the child, N is the number of visits of the parent, and $C = 1.4$ generally.

- When `select` reaches a node that still has an unexplored move, `expand` creates one new child for that move.

- From this child, `rollout` plays a fast random game, but stops after thirty-two half moves; if the position is not finished it scores the board with the same light heuristic used by the Minimax AI.

- The result of the rollout (1 for a win, 0.5 for a draw, 0 for a loss) is then passed to `backpropagate`, which adds the value to wins and increments visits for every node on the path back to the root.

These four steps repeat roughly two thousand times in 5 seconds.

When the timer ends, `performMove()` chooses the root child that has the most visits and calls `game.performMove(x, y)` to play that move on the real Othello board.

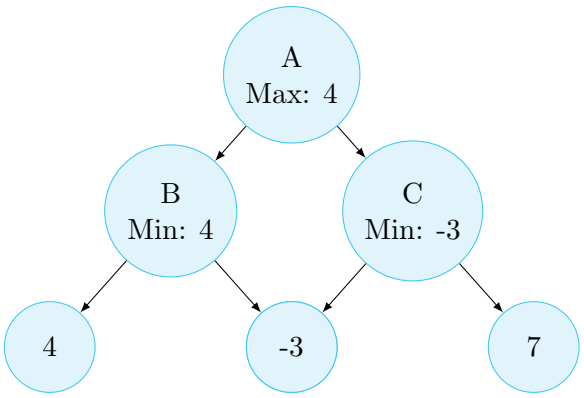
Comparing AIs

AIs (except MCTS) are deterministic. So in order to have relevant statistics, we played 4 - 6 times before letting AIs take over. That method makes the beginning of the game more random and validates that the strongest AI can turn the tide of the game.

Win rate	**Greedy**	**MiniMax**	**MCTS**
Greedy	–	26%	4%
MiniMax	74%	–	28%
MCTS	96%	72%	–

Table 1: Non-deterministic (random) comparison between AIs

This example shows how Minimax works by propagating values from the leaves up to the root.



Annotated Minimax decision tree — values are propagated bottom-up. Max selects B because 4 > -3.

3.3 Voice Control System (Pillar 3)

Overview

The voice control system represents a critical component of our Othello game implementation, enabling players to interact with the board using natural spoken commands. This module enhances both user experience and accessibility by allowing hands-free control of game actions.

Implemented through a dedicated `VoiceController` class, the system integrates the following components:

- **Speech Recognition Engine** – based on the offline Vosk toolkit
- **Audio Processing Pipeline** – handles real-time input capture and preprocessing
- **Command Interpretation** – maps speech to valid Othello coordinates
- **Error Correction** – phonetic mappings for improved robustness

Audio Configuration

The voice module is tuned for real-time audio capture using the following parameters:

- **Input Sample Rate:** 48 kHz (raw microphone capture)
- **Target Sample Rate:** 16 kHz (optimized for speech recognition)
- **Block Size:** 4000 samples
- **Audio Channel:** Mono

Speech Recognition Model

The system uses the Vosk `small-en-us` model, which operates fully offline with low latency and minimal resource usage — an essential requirement to ensure fluidity and responsiveness on the Raspberry Pi.

Coordinate Recognition and Correction

All 64 board positions (from A1 to H8) are pre-generated and stored. Speech is matched against these possible coordinates, with a phonetic correction system to address common misrecognitions:

- **Letter Corrections:**
  - "HEY", "EI" → A
  - "BEE", "BE" → B
  - Similar mappings for C–H
- **Number Corrections:**
  - "WON", "WAN" → 1
  - "TO", "TOO" → 2
  - Complete corrections up to 8
- **Special Cases:**
  - "BEFORE" → B4
  - "SEEFOUR", "SEAFOUR" → C4

This mapping significantly reduces recognition errors by addressing homophones, user accent variation, and ambiguous input.

Real-Time Audio Pipeline

The core of the system lies in the `_callback()` method, which performs the following operations:

1. **Audio Preprocessing** – clipping and normalization
2. **Resampling** – downsample from 48kHz to 16kHz via `librosa`
3. **Format Conversion** – to 16-bit PCM format for Vosk
4. **Recognition** – real-time inference via Vosk recognizer
5. **Correction Mapping** – phonetic correction and standardization
6. **Move Validation** – ensure the corrected command is within valid moves

Error Handling and User Feedback

The system implements robust and user-friendly error handling:

- Invalid or misinterpreted moves trigger retries and feedback
- Keyboard interrupts (Ctrl+C) do not crash the game
- Continuous listening is maintained until a valid move is detected

User Interface Integration

The `listen_for_move()` function orchestrates the voice capture process:

- Starts the audio stream and displays a mic icon to let yhe player that he can speak
- Streams input to the recognizer continuously
- Returns a valid coordinate string (e.g., `"A1"`) or `None` in case of errors

Game Logic Integration

Integration with the Othello engine is seamless:

- Output format is standardized as strings like `"E3"` or `"G7"`
- Errors are propagated via `None`, allowing the main loop to handle retries
- The voice state is maintained between interactions for stability

Performance and Testing

Empirical testing during gameplay shows an average recognition accuracy of approximately **55%**. This rate is affected by environmental noise, pronunciation variation, and inherent limitations of small speech models. As a result, multiple speech attempts are sometimes necessary for a successful command.

Despite these limitations, the system remains usable and practical, demonstrating the feasibility of integrating offline voice control on embedded systems such as the Raspberry Pi.

3.4 Communication Protocol (Pillar 4)

General Overview

Communication occurs over TCP sockets on port `4321`. The server (host player) listens for incoming connections, while the client connects using a known IP address. Once connected, both instances exchange string-formatted messages based on a custom lightweight protocol.

Message Format

Each message is a string following this structure:

`{CODE}|{MOVE}|{PLAYER}`

Arguments are separated by the `|` character. All letters must be uppercase. A trailing pipe is forbidden. Some examples of valid messages include:

- `0` – Game initialization
- `1|C6|B` – Move played at position C6 by Black
- `-3` – Illegal move

Codes are classified as:

- Positive integers — instructions or actions
- Negative integers — acknowledgements or error codes

Code	Meaning	Argument 1	Argument 2
0	Initialize game	—	—
1	Play move	Position (e.g., C6)	Color (B or W)
2	End game	Winner (B, W, or NONE)	—
3	Resignation	—	—
4	Timeout notification	—	—
-1	OK / Acknowledged	—	—
-2	Internal error	—	—
-3	Illegal move	—	—
-5	Malformed message	—	—
-6	Lost move (resent)	Position	Color
-7	Abort game	—	—

Table 2: Message types used in the protocol

Software Architecture Integration

The communication logic is implemented in `connection.py`, while the main game loop is handled by the `GameEngine` class in `engine.py`. During setup:

- The server calls `start_server()` to bind the socket and wait for a connection.

- The client uses `start_client(ip)` to connect and sends an initial `0` message.

The module provides high-level functions such as `send_move()`, `send_timeout()`, `send_abort()`, or `send_game_end()` to abstract message formatting and socket writing.

Typical Game Sequence

Automatic Acknowledgement When a message of type `2` (end of game) is received, the recipient automatically replies with a `-1` message to confirm proper receipt. This behavior is implemented by default in the communication module.

1. **Initialization**: the client sends `0`; the server replies with the first move.
2. **Game phase**: players alternate with messages like `1|XY|P` (XY is position, P is color).
3. **Game end**: one player sends `2|V` (V is the winner), the other replies with `-1`.

Error Handling and Resilience

The protocol includes robust handling for communication failures and invalid states:

- `-3` — Invalid move: wrong turn, illegal position, or wrong color.
- `-5` — Malformed message: syntactic or semantic issues.
- `-6` — Move was sent but not received (e.g., after timeout).
- `-7` — Game aborted after repeated protocol violations.

The code `-6` is used when a player has played a move, but suspects it was not received due to a network issue. Instead of resending it as a regular move (`1`), the player uses `-6|XY|P` to explicitly indicate a lost move. This helps avoid conflicts when the opponent has already sent a timeout.

Each player uses timers (60 seconds per move). If the opponent does not play in time, a `4` (timeout) message is sent followed by a victory declaration. Duplicate errors result in game abortion.

Each message code may trigger specific subcases of errors (e.g., syntax errors, invalid order, timeouts). For clarity, these are abstracted in this report. The full list of scenarios is available in the reference documentation.

Example of a Perfect Game Sequence

```
P2: 0
P1: 1|D3|B
P2: 1|E6|W
P1: 1|C4|B
...
P1: 2|B
P2: -1
```

Logging and Debugging

All print statements are redirected to a log file located at `logs/game_log.txt`, which captures the complete runtime activity of the game. This includes both sent and received messages, system events, and error traces.

4 Environmental and Cost Considerations

4.1 Sustainable and Low-Cost Design

Our project was developed with sustainability in mind. All components—Raspberry Pi, USB microphone, screen—were reused from school-provided material, avoiding any new manufacturing or shipping impact.

The casing was made using recycled cardboard, and no plastic or 3D printing was used, further reducing environmental cost.

4.2 Energy Efficiency

The system runs entirely on a Raspberry Pi 4 (under 5W), with no remote server or cloud processing. Voice recognition, gameplay logic, and AI all run locally, ensuring both low energy consumption and full data privacy.

4.3 Estimated Cost

Though no new components were bought, the estimated cost to reproduce the setup is:

Component	**Estimated Cost**
Raspberry Pi 4	35 €
Microphone	10 €
Screen	15 €
Misc. (cardboard, cables, etc.)	5 €
Total	**65 €**

Table 3: Estimated cost of the hardware components

Thanks to reuse and modularity, the project remains affordable and eco-friendly.

5 Conclusion

5.1 Summary of Achievements

This project successfully delivered a fully functional, voice-controlled, AI-driven Othello game running on Raspberry Pi 4. It features a clean and responsive graphical interface, an offline speech recognition module, a competitive artificial intelligence agent, and a robust TCP-based communication protocol that enables multiplayer gaming between two separate devices.

All four pillars were implemented, tested, and integrated into a single system. The project respects constraints in terms of energy consumption, user privacy, and modularity. In doing so, it demonstrates the feasibility of deploying interactive, intelligent, and connected applications on embedded systems.

5.2 Challenges and Lessons Learned

Throughout the development process, several challenges were encountered:

- **Real-time communication:** Designing a reliable and human-readable network protocol required careful planning and error handling mechanisms to ensure consistent synchronization.
- **Voice control accuracy:** Achieving reliable command recognition with limited computing power and environmental noise was difficult. Phonetic correction mechanisms had to be implemented and tuned manually.
- **Integration of independent modules:** Merging voice, AI, network, and GUI components required well-defined interfaces and robust debugging tools.
- **Resource constraints on Raspberry Pi:** Optimizing performance and avoiding crashes or slowdowns were essential, especially for continuous audio processing and concurrent game logic.

These obstacles taught the team valuable lessons in embedded development, modular software design, and fault-tolerant system architecture. Team collaboration, code versioning, and structured testing were also essential for achieving a working prototype within the given timeframe.

5.3 Potential Improvements

While the current version meets the core functional goals, several enhancements could be explored in future iterations:

- **Improved speech recognition:** Training a custom model on game-specific vocabulary or using hybrid keyword spotting and confirmation could significantly improve recognition accuracy.
- **User feedback and accessibility:** Adding voice feedback, visual move confirmation, or vibration support could make the game more accessible, especially to visually impaired users.
- **More advanced AI:** The current AI uses classic minimax search and Monte Carlo Tree search. Replacing or augmenting it with neronal network or reinforcement learning could improve its strategic depth.
- **Cross-platform multiplayer:** Extending the protocol to support communication over the internet and across different operating systems would broaden the game's reach.

This project not only showcased the students' ability to design, implement, and integrate complex systems, but also provided a realistic environment to experiment with real-world constraints in embedded AI, human-machine interaction, and network programming.

6 User Manual

6.1 Othello Game User Manual

Welcome to H2Othello! This manual will guide you through installation, gameplay, controls, network features, and customization options.

Installation

1. **Requirements:**
   - Python 3.9 or later
   - Dependencies (listed in `requirements.txt`): `pygame`, `pygame-menu`, `numpy`, `librosa`, `sounddevice`, `pyaudio`
2. **Setup:**
   (a) Clone the repository:

   ```
   git clone https://github.com/Mflavien01/H2Othello
   cd H2Othello
   ```

   (b) (Optional) Create a virtual environment:

   ```
   python -m venv monenv
   source monenv/bin/activate  # On Windows: monenv\Scripts\activate
   ```

   (c) Install dependencies:

   ```
   pip install -r requirements.txt
   ```

   (d) Configure microphone (for voice input):

   ```
   Edit line 13 (acquisition frequency) and line 9 (mic_index)
   in ./voice/voice_controller.py if needed.
   ```

Launching the Game

Launch from the terminal with:

```
python main.py
```

Game Modes

- **Single Player:** Play against an AI (Surface, Deep Sea, or Abyss difficulty).
- **Local Multiplayer:** Two players on the same device.
- **Network Multiplayer:** Host or join a game via IP. Uses a custom TCP-based protocol.
- **AI vs AI:** Let two AI players face off automatically.
- **Voice-Controlled Mode:** Say your move (e.g., "D4") by pressing `V` on your turn.

Controls

- **Mouse:** Click to navigate menus and place pieces.
- **Keyboard:**
  - `V` — Voice input (your turn)
  - `R` — Resign the game

Gameplay Instructions

1. Select your preferred game mode from the main menu.
2. For AI matches, choose a difficulty: *Surface* (Greedy), *Deep Sea* (Minimax), *Abyss* (MCTS).
3. In network mode, one player must host, the other joins by IP address.
4. On your turn, click on a valid square or press `V` to use speech (e.g., "F5").
5. The game ends when neither player can move. The player with the most pieces wins.

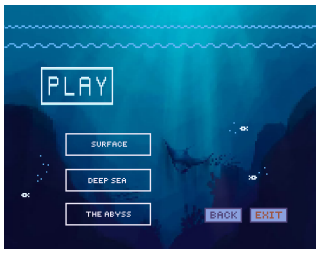
Network Play

- Choose "Network" in the menu and select either "Host" or "Join".
- Enter the opponent's IP address (for Join).
- Uses a custom TCP protocol with codes for move, timeout, resign, etc.

- Handles disconnections and transmission errors gracefully with error codes.

User Interface Previews

Menu



Select Level

Level 1: Surface

Level 2: Deep Sea

Level 3: The Abyss

Customization Options

- **AI Settings:** Configure defaults in `game/config.py`.
- **Visual Theme:** Modify assets in the `images/` folder.
- **Voice Recognition:** Change Vosk model or language under `voice/`.
- **Network Port:** Update the `DEFAULT_PORT` value in `game/config.py`.

Troubleshooting

- Voice input not working? Check the microphone index and sampling rate in `voice/voice_controller.py`.
- Network connection failing? Ensure both players are on the same network or configure port forwarding.
- Logs are saved in `logs/game_log.txt` for debugging.

7 Annex

7.1 Source Code Repository

The full source code of the project, including all components such as the interface, voice control, AI module, and network communication protocol, is available on GitHub at the following URL:

`https://github.com/Mflavien01/H2Othello`