Report on a multimodal interface for chess

Giuseppina Iannotti[1] and Davide Marincione[2]

[1]1938436, iannotti.1938436@studenti.uniroma1.it
[2]1927757, marincione.1927757@studenti.uniroma1.it

I. Introduction

In this report, we present the design, implementation, and testing, of a multimodal interface for playing chess. The interface allows the user to play with either mouse movements, hand gestures or voice commands. The user can switch between the different modalities at any time during the game. It is implemented in `python` and uses the `pygame` library for the audio-visual inteface, `opencv` and `mediapipe` for hand gesture recognition, and the `dragonfly` library for voice commands. The interface has been tested with a group of 12 users, and the results show that the multimodal interface is more engaging and fun to use than a traditional mouse-based interface.

II. Coding, Graphics & Audio

Libraries: A chess program is quite complex to make, and since writing one would've been a project in itself, we have decided to use the `python-chess` library. This choice enabled us to abstract away the complexity of the game, and to focus on making the interface and the interaction. `python-chess` is used all over the project's code, to make checks and queries to display the correct information, but also to interact with the chess engine and to make moves.
For the graphics, audio and event handling, we used the `pygame` library, which is a set of `python` modules designed for writing small video games. It is simple yet very powerful, as it allows to draw shapes, images and text on the screen, to play sounds, and to handle user input. Admittedly, it can be quite slow for medium to large-scale projects, as its main drawback is the graphics rendering, which is done via old-fashioned blitting. But, for a project such as ours, it was more than enough.

Programming paradigm: For all its usefullness, `pygame` only provides basic functionalities and none of the data structures and systems used in writing videogames. Things that, to a certain extent, we needed for our project. So we decided to go for an OOP approach, defining ever more refined objects, building on top of more abstract ones. This tactic proved to be extremely successful, as in the later parts of the project there were a couple of instances in which we needed a new class, and we were able to define it without much hassle.

Basic structure: Inspired by many game engines (Unity, Godot, etc.), our main class is the `Object` class, which is the base class for most of the elements in our program 1. It is defined to be as generic as possible: an object in 2d space which, being in a parent-child hierarchy, can have its absolute position changed by the parent, while maintaining its relative position to it. Internally it has a class-wide `OBJECT_COUNTER` member which is used to give each `Object` its `id`, used to recognize it among the other objects.

Rendering: We then have the `Renderable` class, which inherits from `Object`. And, with it, a `Renderer` class, which is a singleton that keeps track of all the `Renderable` objects and renders them. As such, the `Renderable` class has a `draw()` method, a `set_visible()` method and an `order` attribute, which is used to determine the order in which the objects are rendered. The `Renderer` class has a `step()` method, which is called every frame, and it renders all the `Renderable` objects in the correct order.

Clickables: The `Clickable` class inherits from `Renderable` and is used to define objects that can be clicked (in our simple system, we don't have invisible but clickable objects). Like with `Renderer` and `Renderable`, we have a `Clicker` singleton, that keeps track of all the `Clickable`s. It too has a method called every frame, named `highlight()`, which determines which `Clickable` would be clicked if a "click" event was called. When that happens, its `execute_click()` method is called, which runs

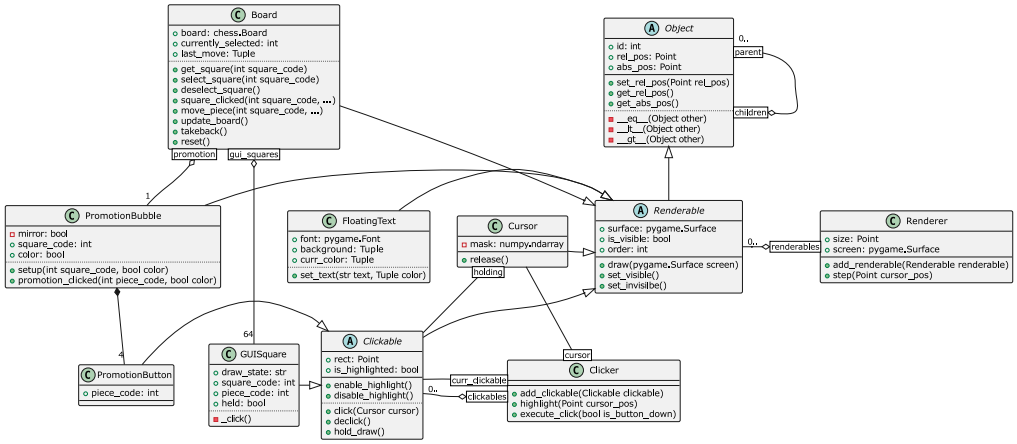

Fig. 1. UML description of the basic structure of the program.

Fig. 2. The cursor, with different backgrounds.

the `click()` and `declick()`[1] methods that each `Clickable` must have. This way, we can define different behaviours for different objects.
Other than that, the `Clickable` class has a `enable_highlight()` method and `disable_highlight()` method, which are used to enable and disable the highlight state of the object, respectively. Furthermore, it has a `hold_draw()` method, used by the `Cursor` class to draw the object when it is being held (if it can be held).

Cursor: The `Cursor` class is a singleton that inherits from `Renderable`. As the name suggests it is used to draw the cursor on the screen, and to keep track of the object that is being held. As described in the next sections it can be driven either by mouse or by hand gestures, and it is referenced by the `Clicker` as it handles which object is being held (as only `Clickables` are holdable). We decided to make the `Cursor` a `Renderable` such as to make it different from the os mouse cursor; we felt that it would be better to have a custom cursor such as to give a distinct visual feedback, especially when the cursor is driven by hand gestures. With this class we have made our first interface choice, as the cursor is designed to be a fairly big cross symbol, which distorts the colors it is drawn upon; making it always visible, no matter the background 2. Initially we had thought of using

$$c^* = 255 - c, \tag{1}$$

but we found this to be insufficient, as one can expect most colors to sit in the middle of the color spectrum, thus getting a situation in which the cursor is invisible on most colors, as

$$255 - 127 = 128. \tag{2}$$

To avoid this, we resorted to

$$c^* = (c + 128) \bmod 256, \tag{3}$$

which assures us to always get a color that is distant enough from the original.

Text: As we sometimes needed to display text on the screen, we have defined the `FloatingText` class, which inherits from `Renderable`. It exploits `pygame`'s `Font` class, which is used to render text on the screen, and thanks to that it can dynamically change the displayed text and the color (as the `Font` class enables that).

Board: The `Board` class is a singleton that inherits from `Renderable`. Internally, it has a `board` attribute, which contains a `chess.Board` object, used to make moves and to check the state of the game. Furthermore, it has a reference to 64 `GUISquare` objects, which inherit from `Clickable` and handle the drawing of pieces on the screen, and their interaction with the user. To

[1]The `declick()` method is run when the left mouse button is released, instead of pressed.

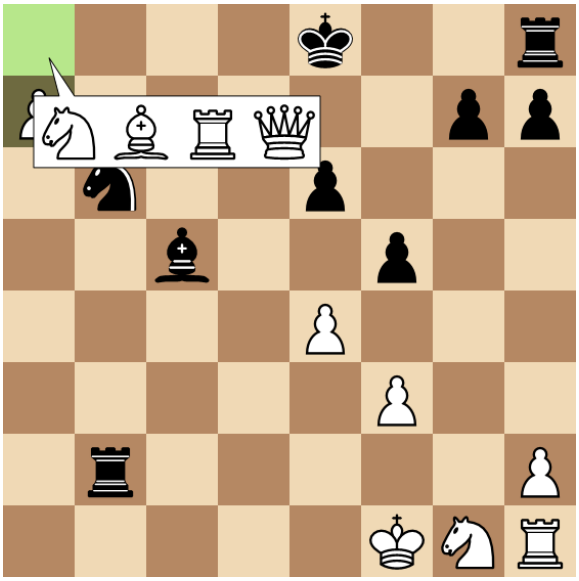

Fig. 3. How the promotion bubble looks like.

build a satisfying interface, `Board` has a number of features:

- It enables a red square under the king when it is in check.
- It marks a square with green when it gets selected (as a move can be done by doing two clicks, the first one to select the piece, the second one to select the destination), and highlights in the same way all of the possible moves.
- It plays a sound when a move is done, and a different one when a king gets in check.

Promotion: The `Board` recognizes when a pawn reaches the last opposite row, when this happens, before executing the move by calling its internal `chess.Board` object, it makes a popup element appear, the `PromotionBubble` 3. This is a `Renderable` which contains four `PromotionButtons`, which inherit from `Clickable`. Each of these buttons represent a piece to which the pawn can be promoted to. When a button is clicked, the `PromotionBubble` calls the `Board` to finalize the move and to promote the pawn.

Game loop, events and AI: The main loop of the program can be found in `chess_main.py`. After the initialization of all the objects and systems, it enters a loop which can be roughly described as follows:

1) Update the cursor position with the latest mouse or hand position, and run `clicker.highlight(cursor_pos)`. If a hand gesture is detected, push a mouse event.
2) Resolve events, such as mouse clicks, key presses (quit game, takebacks), and moves done (for the AI).
3) Resolve voice commands.
4) Run `renderer.step()`.
5) Run metrics recorder.

During the game, the player competes against Stockfish 16 (which, in our tests, we limited); we easily were able to do this thanks to the `python-chess` library, which has an built-in interface to communicate with a multitude of chess engines. The AI is run during event handling, as our `Board` emits a custom `TURN_DONE` event which we use to discern when the AI has to make a move.

Audio: In the game, we only have three different sounds: two of which we already mentioned, played by the `Board`, and the last one played in response to malformed voice commands. The sound played when a move is done is a simple ”wood-on-wood” sound, which reminisces the sound of a piece being moved on a wooden chessboard. The other two sounds, being alarm/attention sounds, are intentionally synthetic beeps, to make them stand out.

Game recording: At the end of development, we wanted to collect quantitative data about the tests. To do this, we implemented a recording system that saves multiple metrics about the runtime; such as the amount of moves made during a modality, the distance traveled with the hand and the number of utterances by the user. In the Experiments section of this report, we'll talk more about this.

III. Hand tracking and gestures

Libraries: The hand tracking and gesture recognition system is enabled by the `opencv` and `mediapipe` libraries. The first is used to capture the video stream of the camera, and the latter is used to detect and track the hand. The rest of the computation (mapping the hand position to the cursor position, detecting gestures, etc.) is done by our code.

Hand detection loop: We design a singleton `HandDetector` class, which runs its own loop on a separate thread with respect to the main system. This is done to avoid blocking the main loop, and to give as smooth an experience as possible. The loop can be described as follows:

1) Capture the video stream.
2) If the video stream has stopped, stop the loop.
3) Flip the video stream horizontally (if needed).

4) If an async call to the `mediapipe` library is not running, execute one with the current frame.

The async call is done to avoid blocking the loop as the `mediapipe` library can be quite slow, and the check for no other async calls is done to not accumulate multiple async calls (which, on a slow laptop, can cause the system to crash). Once the async call gets executed, `mediapipe` runs a custom callback which we design to processes the resulting hand landmarks into the `Hand` class, which we then store in the `HandDetector`.

Normalizing hands: The landmarks $\mathbf{L} \in \mathrm{R}^3$ that `mediapipe` produces live in an approximated screen space. This is fairly useful when mapping the hand to the screen space R^2, as we can just disregard the third dimension. But, when one wants to recognize hand gestures while being invariant to different people's hands, it must be modified. Therefore, before storing them, we normalize the landmarks such as to minimize the differences between different hands, their position and orientation.
First and foremost, we extract the palm center p by calculating a weighted average (handmade by us) over the base of the fingers and the wrist

$$p = \frac{\mathbf{L}_1}{2} + \frac{\mathbf{L}_6 + \mathbf{L}_{10} + \mathbf{L}_{14} + \mathbf{L}_{18}}{8}. \tag{4}$$

We extract the palm width w, as it is invariant to different hand positions,

$$w = ||\mathbf{L}_6 - \mathbf{L}_{18}||_2, \tag{5}$$

and we get the relative landmarks $\overline{\mathbf{L}}$ by normalizing the landmarks with respect to the palm center and the palm width,

$$\overline{\mathbf{L}} = \frac{\mathbf{L} - p}{w}. \tag{6}$$

We could've stopped here, as $\overline{\mathbf{L}}$ is enough to recognize the gestures that we designed for the project; but at the time we weren't sure if we would've needed a more expressive representation for more complex gestures. Because of this, we decided to apply a change of basis to $\overline{\mathbf{L}}$ such as to get $\mathbf{L}^\star$, where the internal palm normal is the z axis, the palm width is the x axis (with the positive direction being that of the pinky, regardless of the handedness), and the y axis as their cross product (which, when calculated, should be the direction of the fingers). To do this, we first get the palm normal

$$\mathbf{n}_{\text{palm}} = (1 - \mathbf{2}_{\text{left}}) \frac{(\mathbf{L}_6 - \mathbf{L}_1) \times (\mathbf{L}_{18} - \mathbf{L}_1)}{||(\mathbf{L}_6 - \mathbf{L}_1) \times (\mathbf{L}_{18} - \mathbf{L}_1)||_2}, \tag{7}$$

then we get the pinky normal,

$$\mathbf{n}_{\text{pinky}} = (1 - \mathbf{2}_{\text{left}}) \frac{\mathbf{n}_{\text{palm}} \times (\mathbf{L}_{10} - \mathbf{L}_1)}{||\mathbf{n}_{\text{palm}} \times (\mathbf{L}_{10} - \mathbf{L}_1)||_2}. \tag{8}$$

and the fingers' normal,

$$\mathbf{n}_{\text{fingers}} = (1 - \mathbf{2}_{\text{left}}) \frac{\mathbf{n}_{\text{pinky}} \times \mathbf{n}_{\text{palm}}}{||\mathbf{n}_{\text{pinky}} \times \mathbf{n}_{\text{palm}}||_2}. \tag{9}$$

This produces a basis $\mathbf{B} = [\mathbf{n}_{\text{pinky}}, \mathbf{n}_{\text{fingers}}, \mathbf{n}_{\text{palm}}]$ that is invariant to the hand's orientation, and that gives explicit meaning to the axes. Finally, we produce $\mathbf{L}^\star$,

$$\mathbf{L}^\star = \overline{\mathbf{L}}\mathbf{B}^T. \tag{10}$$

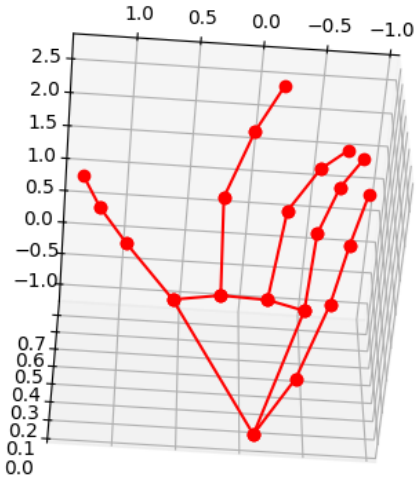

Fig. 4. Right hand in $\mathbf{L}^\star$ doing a grab gesture.

Gesture recognition: At the outset, we wanted to design two different gestures:

1) A grabbing motion, which would let the user drag-n-drop the pieces. Composed when the thumb and the index are close enough to each other.
2) A tapping motion, which would let the user click the squares. Composed by a forward flick of the wrist.

As we were developing the system, we found that the tapping motion was too hard to recognize reliably (as it doesn't require a change in the landmarks' relative positions). At the same time, we found that the grabbing motion we were developing was actually an already established gesture (used in devices like

the Apple Vision Pro) used both for single-click gestures and for drag-n-drop ones. Because of this, we decided to stick with this single gesture, and to use it in a manner akin to pressing the left mouse button.

To recognize the gesture, we design

input `prev_click`
$m \leftarrow ||\mathbf{L}^{\star}_{\text{thumb}} - \mathbf{L}^{\star}_{\text{index}}||_2$
$d \leftarrow \frac{\mathbf{L}^{\star}_{\text{thumb}} \cdot \mathbf{L}^{\star}_{\text{index}}}{||\mathbf{L}^{\star}_{\text{thumb}}||_2 ||\mathbf{L}^{\star}_{\text{index}}||_2}$
if `prev_click` **then**
    return $m < \alpha_\gamma \wedge d > \beta_\gamma$
else
    return $m < \alpha \wedge d > \beta$
end if

where α and β are the normal thresholds, and α_γ and β_γ are the thresholds for the hysteresis effect, when the user is already doing a grabbing motion (to avoid dropping the piece when the user is moving it).

Hand-cursor mapping: As described in the coding section, at each main loop we retrieve the latest hand-cursor mapping, and, if its timestamp is later than the latest mouse timestamp, we use that. We run the hand-cursor mapping during this main loop, using the latest data given by the hand tracking async code. While being fairly precise, `mediapipe`'s hand tracking system is not perfect: many frames can pass between one detection and the other, tracking is noisy and, finally, the system can lose track of the hand. Therefore we had to build a robust mechanism that could handle these issues.

First and foremost, we use the palm p as the point to map the cursor to, and we rescale and clamp its position to a $[m, M]$ space, such that we can make it easier for the user to reach the edges of the window. We do this by

$$r = \text{clip}_{[m,M]}(\frac{p - m}{M - m}). \tag{11}$$

Secondly, we use both the latest position r_t and the previous one r_{t-1} (if these are present, that is). Furthermore, we keep a cursor position c internal to `HandDetector`, which is independent of the cursor position in the main loop. Then we define an algorithm that updates c and returns it, complete with whether a grab or release event happened (whether the user has started or stopped grabbing).

The algorithm is robust in the sense that it tries to avoid experience-ruining events, such as:

1) The cursor jumping around because of noisy tracking, c moves at a constant rate and can either be updated via bilinear or linear interpolation, based on whether r_{t-1} is present, such as to make its movement smooth (at the cost of being a bit less responsive).
2) The cursor moving even when the hand is still. This can either happen because humans have a natural tremor in their hands, which can be picked up by the hand tracking system, or because the tracking is noisy. To avoid this, we only update the cursor if the hand has moved a certain distance.
3) A grabbed piece being dropped for whatever reason, while the user is still keeping a grab gesture. This is avoided by keeping a list of the most recent detections and checking if even one of them is a grab gesture.

With all of this combined we achieved a system that, in spite of all the noise and issues, provides a smooth experience. We describe the tests and the results in the Experiments section.

IV. Voice commands

To enhance user interaction with the application, our project aims to integrate voice commands. Playing chess using verbal instructions can provide a more engaging experience and offer advantages over traditional input methods. In fact, It can be particularly beneficial for users with accessibility needs or those in hands-free environments.

Dragonfly: To implement voice commands, we use the *dragonfly* library, a Python package that facilitates the creation of voice command scripts. Dragonfly allows for the intuitive definition of complex command grammars and simplifies the processing of recognition results by treating speech commands and grammar objects as first-class Python objects. Additionally, It provides a unified front-end interface that seamlessly integrates with Kaldi as a back-end speech recognition engine.

Rules: Vocal instructions are defined using a set of rules, which specify the structure of the commands and associate them with specific actions within the game. Each rule is instantiated as a *CompoundRule* class, which facilitates the straightforward creation of rules based on a single compound specification. This rule class has the following parameters to customize its behavior:

- *spec* : Compound specification for the rules root element

TABLE I
RULES AND SPECIFICATIONS

Rule Name	Specification
Move Rule	"move ([<src_piece>] | [<src_piece> [<prep> <src_square>] to <tgt_square>] | [[<prep>] <src_square> to <tgt_square>]) [and promote to <prm_piece>]"
Capture Rule	"capture (<tgt_piece> [<prep> <tgt_square>] | <tgt_square>) [with (<src_piece> [<prep> <src_square>] | <src_square>)] [and promote to <prm_piece>]"
Promote Rule	"promote [(<src_piece> | <src_square>)] to <prm_piece>"
Castle Rule	"(castle <special_direction> | <special_direction> castle)"
Piece Rule	"<src_piece> ([<prep>] <tgt_square> |in <src_square> <verb> [<prep>] (<tgt_square> | <tgt_piece> [in <tgt_square>]) | <verb> [<prep> <src_square>]([<prep>] <tgt_square> | <tgt_piece> [in <tgt_square>])) [and promote to <prm_piece>]"
Square Rule	"<src_square> <verb> ([<prep>] <tgt_square> | <tgt_piece> [<prep> <tgt_square>])"

- *extras* : Extras elements referenced from the compound spec. It includes choices for prepositions, pieces, and squares.

Grammar: Rules are combined into a Grammar object, which is subsequently loaded into the Dragonfly engine. The grammar object is tasked with processing voice input and comparing it to the established rules. Upon identifying a match, the associated action is executed, such as moving a chess piece or promoting a pawn. Specifically, our Chess Grammar comprises the following set of rules, detailed in I :

- `Move Rule`: Defines a structured syntax pattern for recognizing actions initiated with the verb *"Move"*. Examples of commands recognized by this rule are: "Move E7 to E8 and promote to Queen", "Move Queen on H8 to H7"
- `Capture Rule`: Establishes a syntax pattern for recognizing and processing actions starting with the verb *"Capture"*. This pattern allows for the identification of target and source pieces and squares, along with any promotion involved. Examples of actions recognized by this rule are: "Capture Bishop on E4 with Queen", "Capture Pawn with E6"
- `Promotion Rule`: Specifies the syntax for recognizing and processing pawn promotion actions, beginning with the verb *"Promote"*. It identifies the source piece or square and specifies the promotion piece. Examples of instructions recognized by this rule are: "Promote Pawn to Queen", "Promote to Queen"
- `Castle Rule`: Defines the syntax for recognizing and processing castling commands, focusing on identifying the direction of the castle (kingside or queenside). Examples of commands recognized by this rule are : "Kingside Castle", "Castle Queenside"
- `Piece Rule`: Establishes a structured syntax pattern for interpreting commands starting with a piece name, handling various actions involving a specified piece. Examples of actions recognized by this rule are: "Pawn capture Knight and promote to Queen", "Queen capture Bishop"
- `Square Rule`: Specifies a pattern for recognizing commands that involve specifying source square and, subsequently, a target square or a target piece on the chessboard. This rule is designed to handle actions that involve moving or capturing a piece from a specific square to another square or piece. Example of this commands recognized by this rule are: "A2 move to A3", "E6 capture Bishop on E7"

Each rule contains a method for processing recognition, tasked with managing the identified voice commands. Upon recognition, it extracts the pertinent elements from the outcome and assembles a Command object embodying the chess move. This Command object, encapsulating details such as the action verb, source and target pieces, and their corresponding chessboard squares, serves as a container for the information conveyed by the voice commands. Subsequently, this command is transmitted to the Speech Manager for execution.

Speech Manager: It is responsible for interpreting vocal commands and translating them into actionable moves within the chess application. It

acts as a bridge between the speech recognition system and the chess engine, ensuring seamless interaction between the user's spoken instructions and the game state. Upon receiving vocal commands from the speech recognition system, it analyzes each command to extract relevant information such as the verb (e.g., move, capture), source and target pieces, and destination squares. The function validates the extracted commands against the legal moves available on the chessboard. It filters the available moves based on the provided details and determines the most appropriate move based on the context. Moreover, It handles special cases such as castling and pawn promotion separately to ensure accurate interpretation and execution of these complex maneuvers. To prevent outdated or delayed commands from being executed, the function incorporates a timeout mechanism to discard commands that exceed a specified time threshold. As for the `HandDetector` class, the `Speech Manager` runs on a separate thread with respect to the main system. This is done to avoid blocking the main loop, and ensures efficient processing of vocal commands within the application.

V. Experiments and Results

As previously mentioned, at the end of each game, we aim to collect both qualitative and quantitative data related to it.

Recording System: For this purpose, we implemented a recording system that captures various metrics about the runtime. The collected data, detailed in the JSON file, offers a comprehensive view of the chess game, capturing player moves and AI action. This systematic approach to data collection not only aids in understanding user behavior and preferences but also provides valuable insights into the accuracy and effectiveness of the proposed gameplay modalities.The JSON file is organized into two primary sections, namely the *Player Actions* and the *AI moves*.
Each player action object contains:

- `Action Start` : Timestamp marking the beginning of the action
- `Action Type` : Type of action (e.g., "speech", "hand", "mouse")
- `Utterances` (Optional) : Number of speech utterances if the action type is "speech"
- `Hand Distance` (Optional) : Distance traveled by the hand if the action type is "hand"

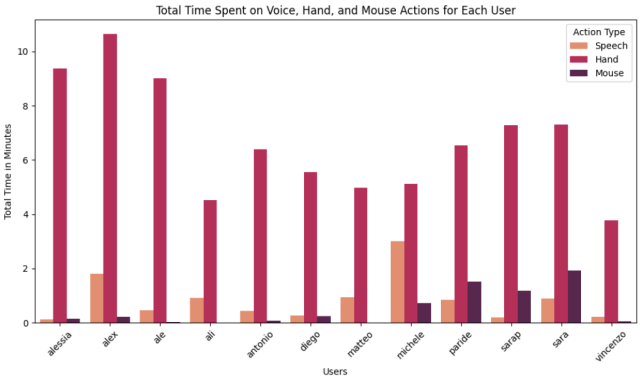

Fig. 5. Time spent on using each modality

- `Mouse Distance` (Optional) : Distance traveled by the mouse if the action type is "mouse"
- `Down and Up Botton` (Optional) : Button press counts if the action type is "hand" or "mouse"
- `Moves` : Array of moves made during the action. Each move is detailed by piece type, starting position, ending position, and any promotion piece, if present
- `Action End` : Timestamp marking the end of the action

Instead, the AI's moves are stored in a list, with each move represented as for Player's Moves.

Metrics: Recording files, containing player's and AI's moves, were processed to extract relevant metrics. We conduct descriptive statistical analyses on our dataset to facilitate more informed interpretations. This involve generating a series of graphs aimed at providing quantitative and qualitative descriptions of specific metrics.
Note : Periods of time during which individuals used to speak without executing any commands were recorded based on the last modality used. Given that the Gesture Modality was the most utilized, the data and distributions reported are influenced accordingly.
`Time Spent Using Each Modality` : In Fig. 5, we illustrate the distribution of time spent by users on each modality. It is calculated by summing the duration of all actions performed using that modality. The plot clearly demonstrates that users predominantly allocate more time to Gesture actions compared to Voice interactions. Concerning the former, no user has reported fatigue using the handtracking mechanism, keeping in mind the tests were usually less than 10 minutes long. On the other side, altough using voice input has garnered particular

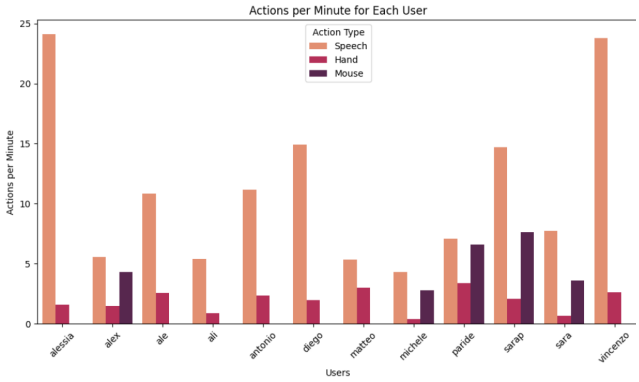

Fig. 6. Number of Actions per Minute performed by users across different modalities

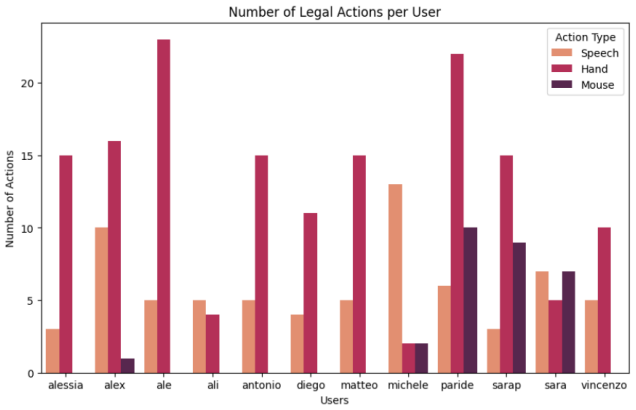

Fig. 8. Number of legal actions performed by users across different modalities

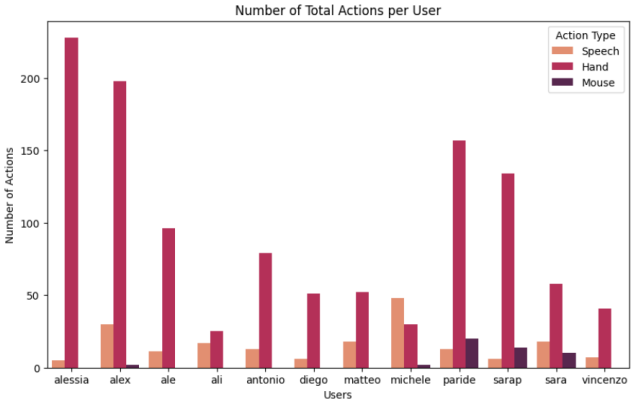

Fig. 7. Number of total actions performed by users across different modalities

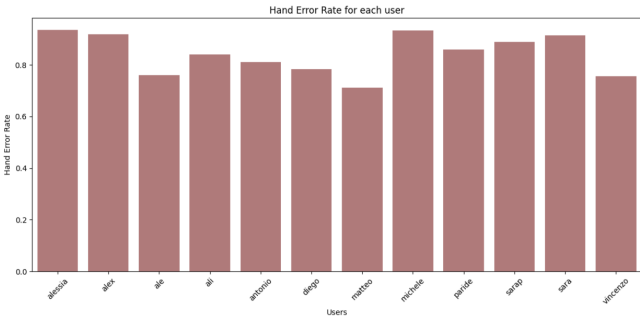

Fig. 9. Hand Error Rate

interest too, its use was limited. This was due both to the greater familiarity with playing by hand and to linguistic difficulties encountered by some users. Instead, only a few users have opted to utilize the mouse as an input modality. Its limited usage may be attributed to users' curiosity and inclination to explore the functioning of alternative modalities. However, in the few cases in which the users chose to use the mouse, they were able to pick it up instantly and without any issues. This reassures us of the solid baseline implemented.

`Actions Per Minute` : Fig. 6 illustrates the frequency of actions performed by users across the different modalities during chess gameplay. It is a measurement of how efficient and fast users perform commands using the different modalities. We can see that the highest bars are those representing speech interactions. However, this distribution is skewed due to the aforementioned **Note**.

`Total Actions` : Fig. 7 illustrates the total number of actions performed by users across different recordings, categorized by their type of modality. The plot indicates that the number of actions performed by users is relatively high, including both legal and illegal commands. As shown, the majority of users opted for gesture-based interactions, being consistent with Fig. 5, in which we see that users have spent more time using the gesture modality.

`Legal Actions` : Fig. 8 shows the distribution of legal actions performed by users across different recordings, categorized by their type of modality.

From the previous plots, we can understand the number of legal actions is relatively small compared to the total number of actions performed. For this purpose, we are particularly interested in analyzing the error rate of the introduced modalities, computed as:

$$1 - \frac{\text{total\_legal\_actions}}{\text{total\_actions}} \tag{12}$$

`Gesture Error Rate` : At the beginning of our experiments, there were issues related to the quality of the hand input, see Fig. 9. Specifically, the piece grabbing was too sticky. This problem

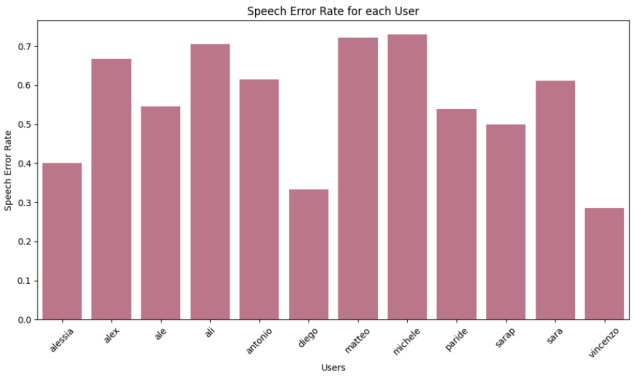

Fig. 10. Speech Error Rate

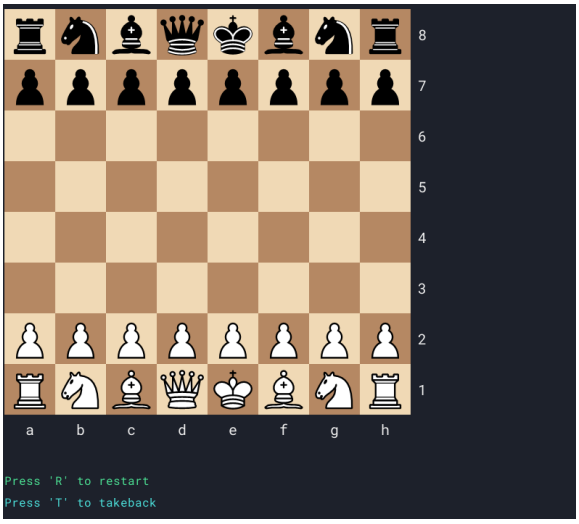

Fig. 11. Final Image of the Interface at the end of the development

was later resolved by removing the hysteresis effect. Additionally, we observed that no user complained about the visibility of the cursor, which we consider a positive indication of its design. As anticipated, it was unnecessary to match hand and cursor in a 1:1 mapping, as users focused more on the cursor's position rather than their hand's position. `Speech Error Rate` : Several factors influences the speech error rate, shown at 10, including users' accents and pronunciation. It is important to note that the implemented grammars cover only a subset of all possible phrases that can be formulated in a chess game. However, this limitation did not pose a significant problem, as the phrases spoken by the users were within the scope of the implemented rules. Additionally, during the experiments, it was observed that the speech recognizer often struggled to distinguish between 'f' and 'e' and 'a'. To avoid comprehension errors, we implemented also the NATO phonetic alphabet, a system that assigns standardized code words to each letter of the alphabet, like 'Alpha' for 'A', 'Bravo' for 'B' and so on. Despite few players used it, it is highly recommended. Moreover, as previously mentioned, many users used to talk while switching between modalities. Their conversations about matters unrelated to the application were often recognized as commands, which, being illegal, triggered a warning sound. This aspect was less appreciated by our users. However, it is assumed that during a chess game, players generally do not engage in unrelated conversations.

Game Over: During the experiments, the need to introduce a "takeback" feature was highlighted. This feature allows users to revert to the previous action, proving especially useful when the gesture or voice recognizer inadvertently processes an unintended action due to a potential misinterpretation. Users report also that the end of the game status was not clear enough, as they were not paying attention to the initial "Press R to restart" message. This led to the implementation of a more visible game over message at the end of the game directly on the board. The interface at the end of the development is shown at Fig. 11

VI. Conclusions

The application provides a novel and interactive way to play chess, offering users the flexibility to choose their preferred input modality. Overall, the application garnered significant interest from the users, providing them with an enjoyable experience. However, despite being the least utilized input method, the mouse remains the preferred mode due to its convenience and suitability for various situations. The collected results indicate potential, but it is important to note that the calculated metrics are biased by several factors, including the limited amount of data collected, lengthy conversations unrelated to chess during the game, and the users' accents and pronunciation. Looking ahead, future work could involve further refining the hand tracking and gesture recognition system, as well as expanding the range of voice commands supported by the application and using a more robust speech recognizer to improve accuracy and reliability. Additionally, one future project might involve implementing chess gameplay through eye-gaze tracking, trying to add a novel dimension to digital chess gameplay.