

# Some title for a multimodal chess interface

Giuseppina Iannotti, and Davide Marincione

*Abstract*—This is the abstract.

## I. INTRODUCTION

This intro has to change!

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 interface, `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 usefulness, `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 `Clickables`. 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

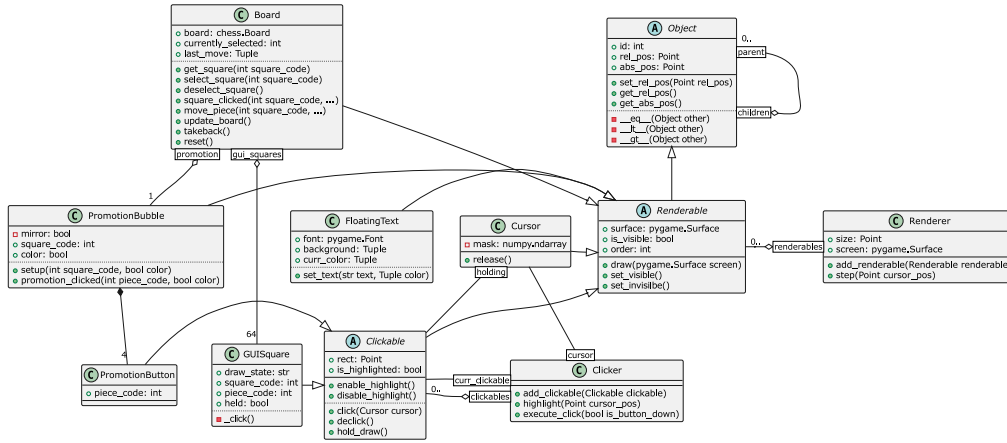


Fig. 1. UML description of the basic structure of the program.



Fig. 2. The cursor, with different backgrounds.

`execute_click()` method is called, which runs the `click()` and `declick()`<sup>1</sup> 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

<sup>1</sup>The `declick()` method is run when the left mouse button is released, instead of pressed.

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

$$c^* = 255 - c, \quad (1)$$

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

$$255 - 127 = 128. \quad (2)$$

To avoid this, we resorted to

$$c^* = (c + 128) \bmod 256, \quad (3)$$

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



Fig. 3. How the promotion bubble looks like.

the screen, and their interaction with the user. To 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.

*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 GESTURES

### IV. VOICE COMMANDS

### V. EXPERIMENTS AND RESULTS

On a note, in each of our experiments, we have found that no user has ever complained about the visibility of the cursor, which we take as a sign that its conception was successful.

### VI. CONCLUSIONS