
# Computer programming project report

**Author:** Piotr Copek

**Date:** April 8, 2025

**Teacher:** dr inż. Michał Piela

# Table of Contents

# Chess

This project involves developing a fully functional chess game using Python and the customtkinter library for the graphical interface. The game follows standard chess rules, including special moves like castling, en passant, and pawn promotion, while providing a customizable user experience through themes, fonts, and colors.

## Key Specifications

1. **Game Logic**
   - **Move Validation**: Enforces standard chess rules.
   - **Special Moves**:
     - Castling (kingside/queenside).
     - En passant captures.
     - Pawn promotion (to Queen/Rook/Bishop/Knight).
     - Two squares at first pawn move.
   - **Game State Detection**: Auto-check for checkmate/stalemate.

   Game logic was built with accuracy and extensibility in mind. All special rules and conditions are tested. The system separates logic from visuals, which simplifies testing and debugging. Handling edge cases like en passant or castling required attention to details and showcases deeper understanding of chess rules.
2. **User Interface**
   - **Interactive Board**:
     - Legal moves highlighted on click.
     - Visual feedback for captures/checks.
   - **Audio Feedback**:
     - Play sound effects for moves.
   - **Move History**: Logs moves in algebraic notation.
   - **Notifications**: Displays alerts for game endings and information about successfulness of applying changes, saving games and loading them.
   - **Audio**: Sound effects for moves/captures (threaded playback).

   In GUI applications it really important to keep everything simple but yet handling complex behaviors. User has to know where and how to perform specific actions. To ensure that user interface is clear I tested app among my friends to receive users feedback.

3. **Customization**
   - **Themes**: Swap piece/assets styles.
   - **Color Picker**: Adjust colors via `Settings`.
   - **Fonts**: Supports custom `.ttf` fonts.

   To create more unique application I decided to add a lot of customization options. The way of implementing fully customizable app is also great way to improve in writing software that is really flexible and easy to maintain as everything has to be centralized and ready to be reused in different parts of the codebase.
4. **Technical Features**
   - **OOP Design**:
     - `Board` (2D grid of `Cell` objects).
     - `Piece` hierarchy (Pawn, Knight, ...).
   - **Modularity**:
     - Separated GUI (`customtkinter`) from game logic.
     - Configurable via `config.ini` and `Settings` menu.

   Object oriented programming is key to create a reuseable and flexible software. Allows to easily create changes during development. Modularity of the project allows to easily add new features and modify already existing logic or features.
5. **Dependencies**
   - Python 3.x, `customtkinter`, `Pillow`, `sounddevice`, `soundfile`.

   All dependencies were carefully selected to match the goals of the project. Lightweight and modern libraries like `customtkinter` help keep the interface clean and responsive, while `sounddevice` ensures seamless audio experience. The project avoids unnecessary bloat, making it easy to run and maintain.

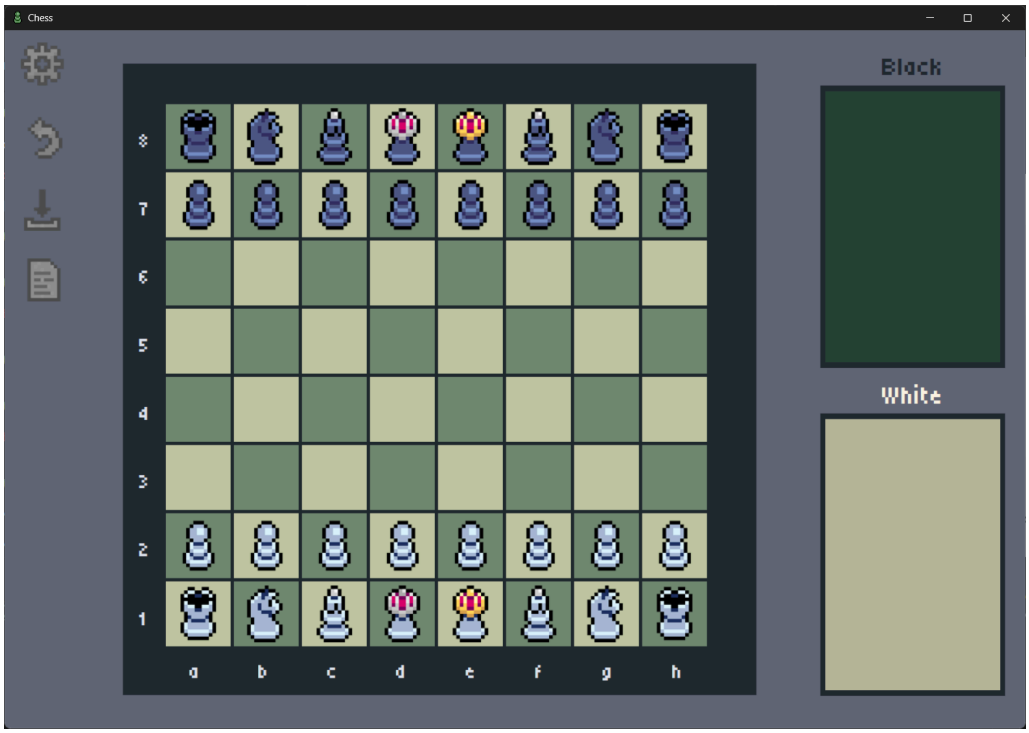
# User manual

## Running from source code

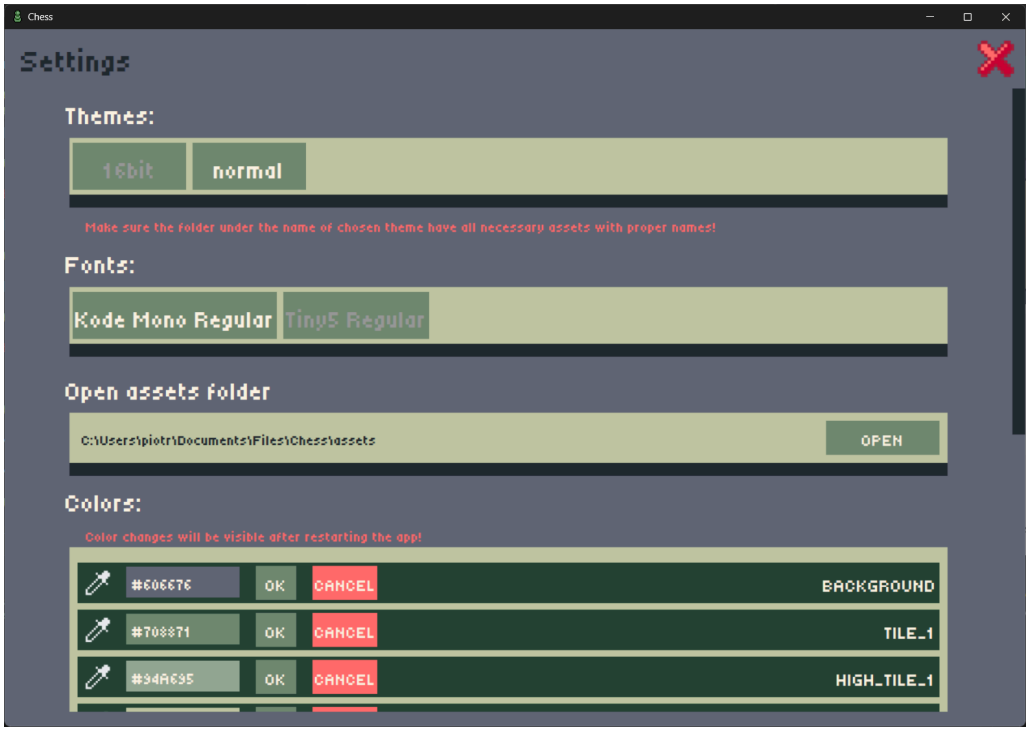
1. Clone repository
2. Navigate to project directory
3. Creating virtual environment is recommended. `python -m venv .venv`
4. Install dependencies `pip install -r requirements.txt`
5. Run application `python .\src\main.py`

# Execution examples

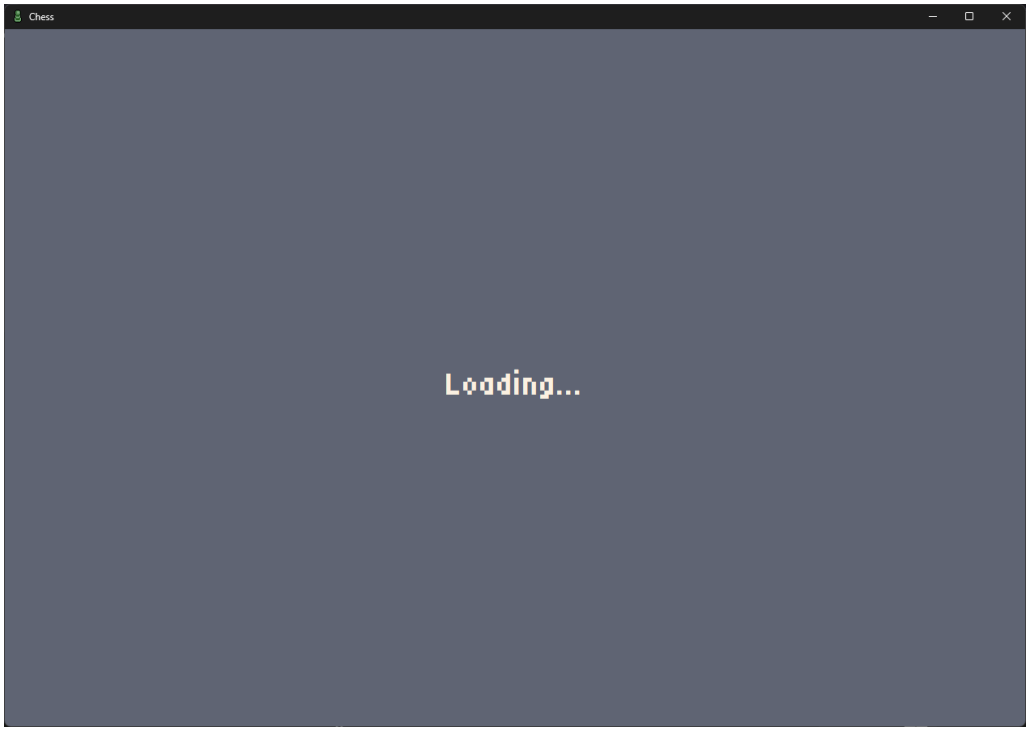
## Main Chess Board




## Settings Menu - Themes and Fonts

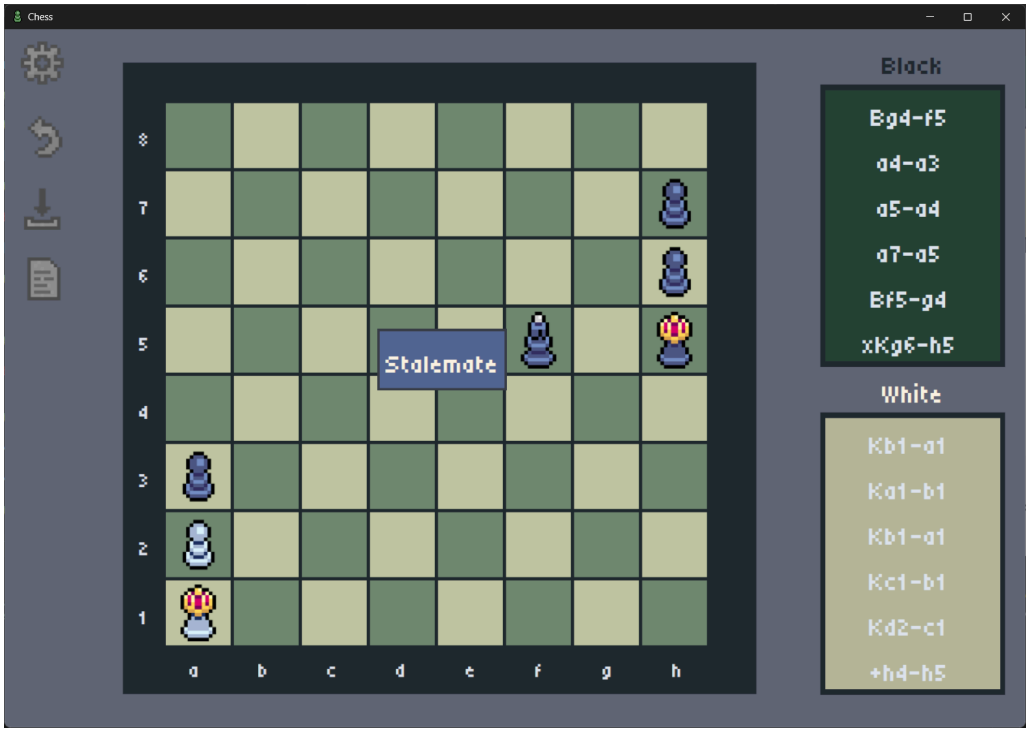


## Loading Screen



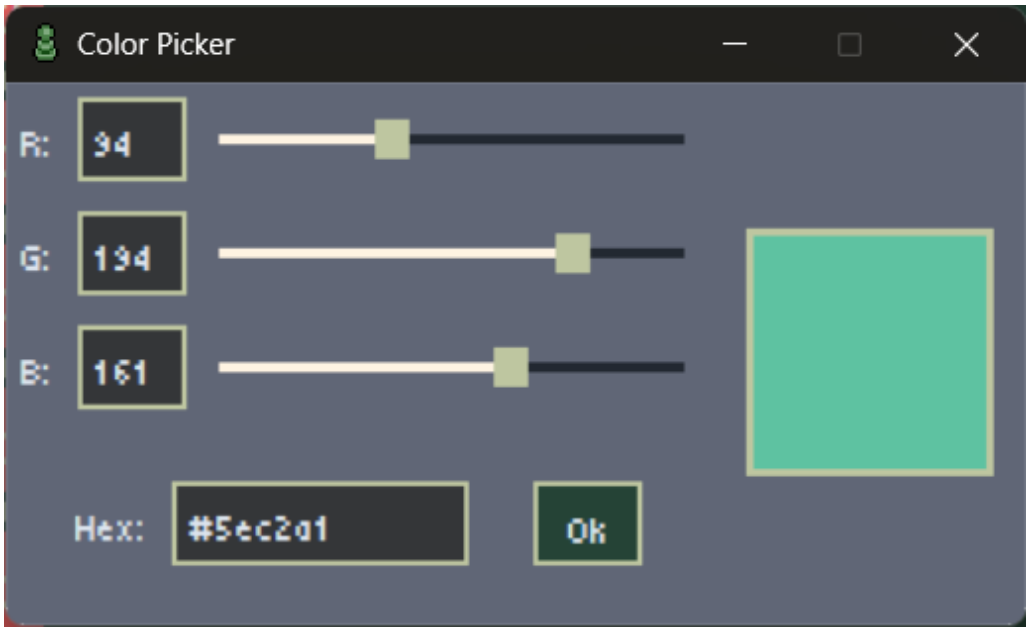

## Checkmate

## Stalemate




## Saves screen

## Color picker




## Saving pop-up

# Internal specifications

## Classes

All classes play key role in app.

### MovesRecord(ctk.CTkFrame)

Class handling recording the moves during playtime. Class stores both players moves in lists and displays notation in two boxes dedicated for each player.

**Relationship with other classes:**

- used by `Board` to pass the current move to it.

**Important methods:**

- `__init__` - properly loads creates menu with clickable elements.
- `record_move(self, moved_piece: Piece, previous_coords: tuple[int, int] | None=None, capture: bool=False, castle: str | None=None, check: bool=False, checkmate: bool=False, promotion: str='') -> None` - creates proper notation for the move and places it in correct frame.

### Saves(ctk.CTkFrame)

Class handles saving, loading and parsing the saves. Saves are located in `saves` directory and are created in `.json` extension for easy readability and allows for creating custom saves.

**Relationship with other classes:**

- Used in `Options` which is class handling opening all user available menus and options.
- Inherits from `CTkFrame` which is widget frame from `customtkinter`.

**Important methods:**

- `save_game_to_file(board) -> bool` - saves game to `.json` file.
- `show_all_saves(self, board) -> None` - shows all saves in dedicated widget.
- `load_save(self, event: Any, board, file_name: str) -> None` - helper function calling all necessary functions to load the game from save. Notifications will indicate if it was successful or not.

## Options(ctk.CTkFrame)

Class displaying all options available for the user:

- settings
- restart game
- save game
- load game from save

Handles interactions with them and closing them as well.

**Relationship with other classes:**

- Invokes `Saves`, `Settings`
- Uses `Board` object to access they methods

**Important methods:**

- `__init__` - properly loads and creates menu with clickable elements.
- `setting_button(self) -> None` - setups setting button. After clicking setting menu will be displayed.
- `replay_button(self) -> None` - button linked to `Board` object invoking `restart` method from `Board` to restart game.
- `save_button(self) -> None` - setups save button. After clicking save menu will be displayed.
- `load_saves_button(self) -> None` - setups saves menu button. After clicking saves menu will be displayed.

## Settings(ctk.CTkFrame)

Class handling changes in user interface. user have option to change assets, colors, and fonts.

Application allows custom assets, but they have to be named properly, and fonts with only `.ttf` extension. In setting all option will be displayed as buttons which will handle the change of the assets and saving the changes to the config file.

**Relationship with other classes:**

- Invokes `ColorPicker` after clicking color pipet in customization menu
- Used by `Options`

**Important methods:**

- `__init__` - properly loads and creates menu with clickable elements.
- `list_directories_os(path: str) -> list[str]` - function displaying all directories. Function acts as scanner for themes and fonts in `assets` folder.
- `select_theme(self, choice: str, button: ctk.CTkButton) -> None` - saves user chose theme of the figures.
- `open_file_explorer(path: str) -> None` - opens file explorer foe easy access to `assets` folder.
- `select_font(self, font: str, button: ctk.CTkButton) -> None` - saves user chose font to config file.
- validation methods to validate color values, length of user input.
- `change_colors(self) -> None` - saves user chose color to config file.

## SaveName(ctk.CTkTopLevel)

Input dialog handling naming the save file. Allows user to cancel the saving procedure. Class is taking care of stopping the main window from user interactions.

**Relationship with other classes:**

- used by `Saves` after saving game was invoked.

**Important methods:**

- `__init__` - properly loads and creates menu with clickable elements.
- `get_save_name(self) -> str | None | bool` - getter for user input from the entry widget, returns string if name is valid, None if user decides to keep default save name and bool if canceled with closing window with ❌.
- `on_save_button(self) -> None` - function validating user input for save name.

## Notification(ctk.CTkFrame)

Class used for displaying user friendly messages with errors and game information. Frame have appearing and disappearing animation for better user experience.

**Relationship with other classes:**

- used all across the application (eg. `Board`, `MainWindow`)

**Important methods:**

- `__init__` - properly loads and creates notification with clickable elements.
- `show_notification(self) -> None` - shows the notification on the screen

## ColorPicker(ctk.CTkTopLevel)

Custom color picker with rgb values sliders and entires, as well with hex entry. Provides live preview of the selected color with values preview as well. User can accept the new color or cancel the procedure of changing the color.

**Relationship with other classes:**

- used by `Settings` fo easy customization fo colors in app.

**Important methods:**

- `__init__` - properly loads and creates menu with clickable elements.
- `color_preview(self) -> None` - creates preview widget for selected color
- `validate_hex_color(value_if_allowed: str) -> bool` validates user input to match hex color encoding.
- `validate_input(P: str) -> bool` - validates user input on every ey release.
- `convert_to_hex(self) -> str` - converts RGB to hex.
- `convert_to_r_g_b(self) -> tuple[int, int , int]` - converts hex to RGB.
- `get_color(self) -> str | None` - returns valid hex value after closing the color picker widget if selected color is valid `None` otherwise.

## COLOR(StrEnum)

Enum holding all colors form config file. To avoid unpleasant long wait time of changing the color of each widget the changes will only be visible after restarting the app with new colors loaded in enum.

**Relationship with other classes:**

- used by all components of the application.

**Important methods:**

- enum has no methods itself but is loaded using custom `StrEnum` from dictionary taken from `config.ini` using `get_colors() -> dict`.

```
def create_color_enum():
    colors = get_colors()
    capitalized_colors = {key.upper(): value for key, value in colors.items()}
    return StrEnum('COLOR', capitalized_colors)
```

## Piece

Abstract class with general implementation of the figures. Holds the asset image, color and important flags. Ensures that the child class have implementation to handle moves. For debugging purposes the `__str__` function is overwritten with custom string:

**Relationship with other classes:**

- Parent class for every figure.

**Important methods:**

- `__init__` - loads asset for the piece and setups all important data for the figure.
- `check_possible_moves(self, color: str, checking: bool=False)` - abstract method
- `check_turn(self, current_color: str) -> bool` - checks current color of figures to move.
- `load_image(self, piece: str | None=None) -> None | ctk.CTkImage` - loads correct image from `assets\{name_of_the_current_theme}` directory.
- `update_image(self) -> None` - updates image of the figure (eg. used for en passant).
- `__str__(self) -> str` - string representation used for debugging.

## Cell

Class handling actions inside specific cell and linking figure to the position on the board.

**Relationship with other classes:**

- Used in `Board`. Each cell in 2D list is an unique `Cell` object links the position on the board to the figure.
- `Cell` holds `Piece` inside with information of which figure is in this position. Might be `None`.

**Important methods:**

- `__init__` - holds position info, figure info, and board reference.
- `on_click(self, event: Any) -> None` - handles clicks on board and calls appropriate functions from `Board`.
- `update` - updates figure asset in cell

## Board

Class handling all cells and move related logic.

**Relationship with other classes:**

- Uses `Cell` in 2D list which links positions and figures on the board.
- Used in `MainWindow`

**Important methods:**

- `__init__` - loads all important data, setups board and flags.
- `determine_tile_color(pos: tuple[int, int]) -> str` - determines color of the cell.
- `create_board(self) -> list[list[Cell]]` - creates representation of the board and returns it.
- `is_game_over(self) -> tuple[bool, bool]` - checks if game is over and how it ended.
- `handle_clicks(self, figure: piece.Piece, position: tuple[int, int]) -> None` - handler for the clicks before figure is selected.
- `handle_move(self, position: tuple[int, int]) -> None` - handler for moving the pieces.
- `check_check(self, move_from: tuple[int, int], move_to: tuple[int, int]) -> bool` - checks for check of the king.
- `handle_game_over(self, in_check: bool, promotion: bool, capture: bool, check: bool) -> None` - decides what to do if game is over.
- `restart_game(self) -> None`
- `load_board_from_file(self, file_info: dict) -> bool` - loads board from parsed information from config file.

# Technics from classes

During development, I applied some of programming techniques from the course.

## Regex

Regular expressions is powerful tool used to validate user input. It was used all across the project, almost for every user input `re` library which is python version of `regex` is used. It ensures that code will properly handle user input and wont cause errors or bugs in application.

## File system

For loading gam assets I used `os` library. It has `path` module which have all necessary functions to handle all file and directories actions. Library is used to get absolute path to assets which is crucial for independently of device proper loading and using files or directories. Library is also used to independently of operating system join paths - instead of hardcoding paths `assets\\pawn_w.png` which is windows specific i used `os.path.join('assets', 'pawn_w.png)`.

## Threading

For better user experience it's crucial to have smooth and not freezing window. Threading ensures seamless transitions through menus and loading assets. Threading is also used to create in-app animations. Threading ensures that window wont freeze during changing assets/ fonts, performing algorithms or playing sound.

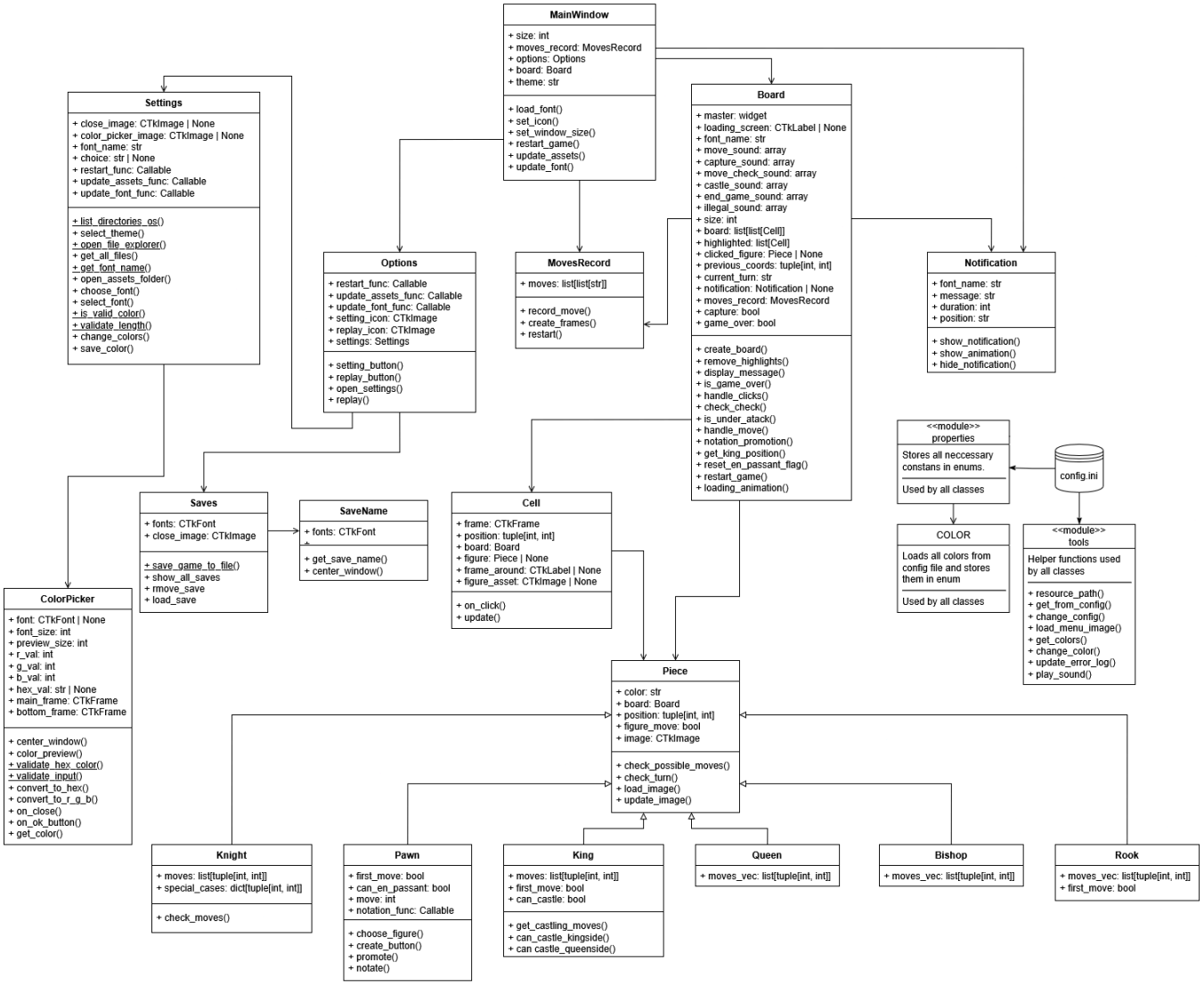
## Modules

Modules are crucial part of python. To keep the project organized and readable, I divided the code into multiple modules. Each module was responsible for a specific part of the application like handling figures, saving the game, or managing the UI. This approach improves structure, makes debugging easier, and allows for faster development.

## Ranges

In many parts of the game logic, `range()` was used for iterating over board positions and generating possible moves. It is especially useful in chess when checking rows, columns, or diagonals step by step. Using `range()` allowed me to control iterations precisely, whether it was looping forwards, backwards, or skipping certain positions. It made movement calculation for pieces like rooks, bishops, and queens easier to implement and understand.

# Class hierarchy



# Algorithms

To fully implement chess game uses of algorithms is necessary. Path finding, checking at least one move forward to ensure king safety and more.

## Legal moves generator/checker

Each piece has its own implementation of checking possible moves. Another function checks if this move will be legal - which means that player wont put their own king at risk of check. From all possible moves generated by figure algorithm some squares might be sieved to ensure legality of the taken move.

## Path finding

Figures move uses something similar to raycasting to find paths that are not blocked by other figures. All paths are hardcoded as a 2D vector acting as direction to check. Direction is checked until some figure steps on its way or there is the end of the board.

## End of game detection

Game will check every time if checkmate or stalemate occurred by analyzing if there exist such move that can be performed bu current player. If there is no legal move game will check if king is in check. If is then it is checkmate otherwise it's stalemate.

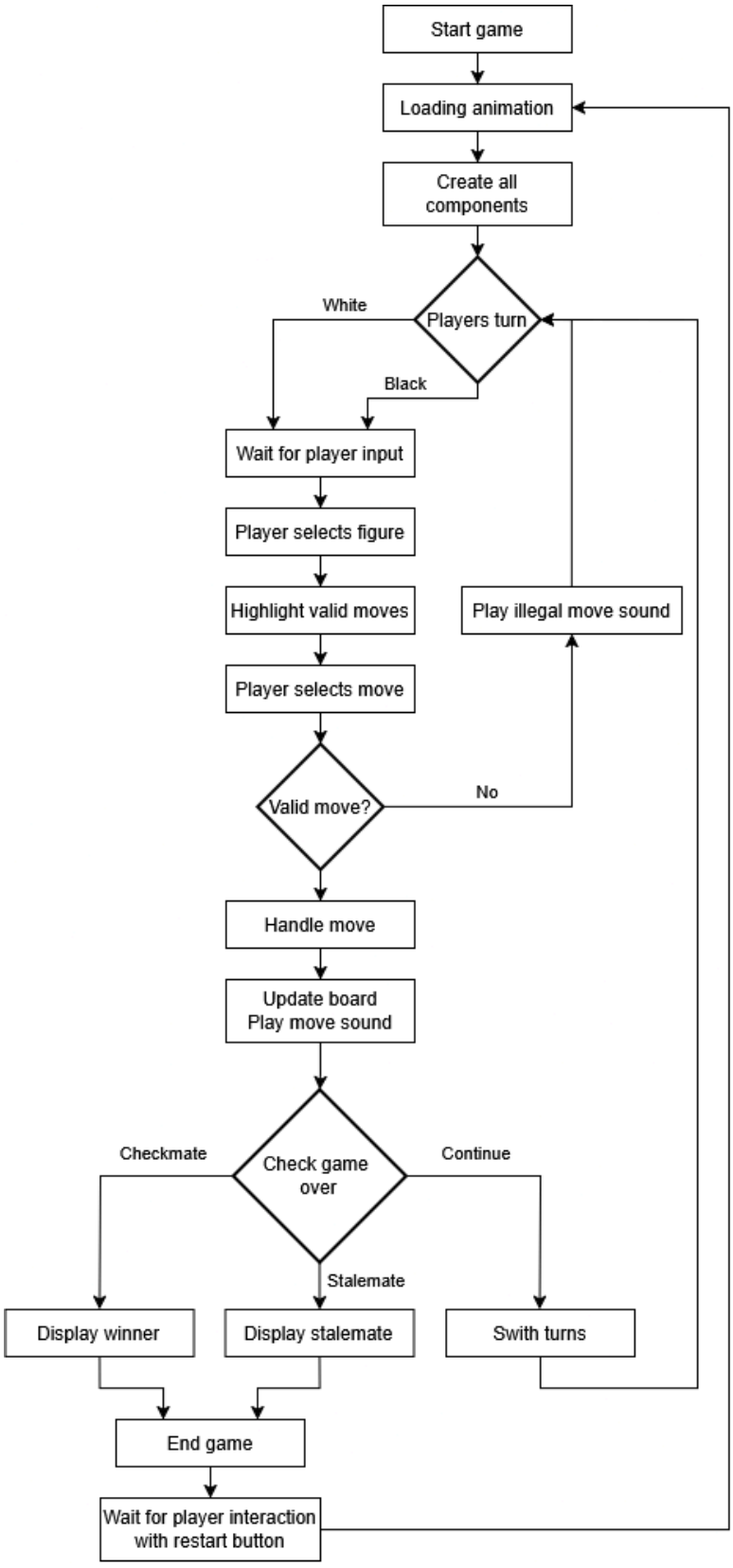
## Validations of moves

- castle - validates if castle and king didn't move before and any of the squares that could be used to castle aren't under attack.
- en passant - checks if the pawn that might be taken was taking first move.

## Promotion

Checks if pawn is in the last possible row for its color. If this situation occurs the widget with figures available will pop-up forcing player to choose one of four figures (Knight, Bishop, Rook or Queen).

# Program operation



# Testing

Due to the complexity of the project, testing was done continuously during the implementation phase. Each component was checked for edge cases and unexpected behavior. While developing, several bugs were discovered that required deeper debugging and sometimes refactoring core logic. Below are some examples of specific issues that were encountered and how they were resolved.

## Pawns problem

Each figure was tested for edge cases. During testing there was a bug with pawn. Pawn couldn't cause checkmate even tho it was clearly a checkmate. To resolve the bug I had to rewrite the function checking the game over and check.

```
def is_game_over(self) -> tuple[bool, bool]:
    in_check = False
    for row in self.board:
        for cell in row:
            if cell.figure and cell.figure.color == self.current_turn:
                possible_moves = cell.figure.check_possible_moves(self.current_turn)
                for move in possible_moves:
                    if not self.check_check(cell.figure.position, move):
                        return False, False
    for row in self.board:
        for cell in row:
            if isinstance(cell.figure, piece.King) and (
                    cell.figure.color == self.current_turn
                ):
                if self.check_check(cell.figure.position, cell.figure.position):
                    in_check = True
                    break
    return True, in_check
```

Faulty check for game over was causing problems with pawns. To fix it i had to rewrite the function to fit proper handling of checkmating with pawn.

```
def is_game_over(self) -> tuple[bool, bool]:
    in_check = False
    has_legal_moves = False
    for row in self.board:
        for cell in row:
            if cell.figure and cell.figure.color == self.current_turn:
                possible_moves = cell.figure.check_possible_moves(self.current_turn)
                for move in possible_moves:
                    if not self.check_check(cell.figure.position, move):
                        has_legal_moves = True
                        break
                if has_legal_moves:
                    break
        if has_legal_moves:
            break
    king_position = self.get_king_position(self.current_turn)
    in_check = self.is_under_attack(king_position, self.current_turn)
    return (not has_legal_moves, in_check)
```

After rewriting function, everything was working properly including checkmating with pawn.

## Threads problem

There was one overuse of threads that caused bug forcing user to close the application almost every two tries of saving the game. The bug was annoying to find due to its randomness. But after testing I found out that creating `TopLevel` window in `customtkinter` library on separate thread was causing misbehavior on the main thread as not everything was handled properly. After all it was really easy to fix but annoying to identify faulty code.

```
def save_game(self, event: Any) -> None:
    threading.Thread(
        target=Saves.save_game_to_file, args=(self.get_board_func(),)
    ).start()
```

Thread was sometimes destroying `TopLevel` window due to additional click on main app window so i moved the task to the main thread and also started checking if saving the game was successful.

```
def save_game(self, event: Any) -> None:
    if Saves.save_game_to_file(self.get_board_func()):
        Notification(self.master, 'Save was created successfully', 2, 'top')
```

## Color picker widget

There was a minor oversight with updating hex value on changing RGB values in entries. This required binding converting to hex function to `<KeyRelease>` action.

```
self.r_val_label.bind('<KeyRelease>', lambda e: self.update_hex(e))
self.g_val_label.bind('<KeyRelease>', lambda e: self.update_hex(e))
self.b_val_label.bind('<KeyRelease>', lambda e: self.update_hex(e))
```

```
def update_hex(self, event=None) -> None:
    self.hex_val_label.delete(0, ctk.END)
    self.hex_val_label.insert(0, f'{self.convert_to_hex()}')

def convert_to_hex(self) -> str:
    return f'#{self.r_val:02x}{self.g_val:02x}{self.b_val:02x}'
```

This simple fix resolved the issue.

## GUI related bugs

Sometimes i forgot to check if specific frame actually exists. This was causing some `customtkinter` errors. To resolve I started using type annotations with `mypy` static checker. This helped with proper handling destroying and creating frames. These fixes was quite easy and only required `if` `else` logic blocks.

Simplified example of bug:

```
import customtkinter as ctk

widget = None
widget.pack()
```

This can be rewritten in such way that is safe from bugs:

```
import customtkinter as ctk

widget: ctk.CTkFrame | None = None
if widget:
    widget.pack()
else:
    widget = ctk.CTkFrame(master)
```

## Incorrect en passant handling

During final phase of writing application I started refactoring the code with optimizations and cleanness in mind. Unfortunately by mistake I made additional indent to the function responsible for resetting flag that indicates that pawn can be taken in this special move.

```
if isinstance(cell.figure, piece.Pawn):
    if cell.figure.first_move and abs(self.previous_coords[0] - row) == 2:
        cell.figure.moved_by_two = True
    else:
        cell.figure.moved_by_two = False
    if cell.figure.promote():
        promotion = True
    self.reset_en_passant_flags(cell.figure.color)
```

One additional tab made big difference as user could choose any other figure to move and `en_passant` flag wouldn't reset. After analyzing refactored code i quickly found the issue using some debug `print` statements.

```
if isinstance(cell.figure, piece.Pawn):
    if cell.figure.first_move and abs(self.previous_coords[0] - row) == 2:
        cell.figure.moved_by_two = True
    else:
        cell.figure.moved_by_two = False
    if cell.figure.promote():
        promotion = True
self.reset_en_passant_flags(cell.figure.color)
```

### Other minor bugs

During the development process, I encountered many smaller bugs and issues, such as typos, forgetting to call newly implemented methods, or skipping small but necessary adjustments after adding new features. These kinds of human errors are quite common, especially when working on a larger codebase. Fortunately, Python’s clear and descriptive error messages made them easy to spot and debug quickly. Most of these problems were resolved within minutes, and often helped highlight areas where the code could be made more robust or better organized.

# Conclusions

The chess project provided a valuable opportunity to apply object-oriented programming principles in a complex, interactive application. Developing both the game logic and user interface involved addressing various challenges, from implementing accurate move validation and special chess rules, to ensuring a smooth and responsive user experience through the use of customtkinter. The structure of the code evolved through extensive testing, which led to improved clarity, reusability, and correctness. By solving issues such as faulty checkmate detection and GUI instability caused by threading, the final project became not only functional but also reliable. The project also reinforced the importance of thorough debugging, consistent coding practices, and maintaining clear separation between logic and interface.

# Documentation

Documentation was created using `pdoc` tool. Full documentation is available under this link chess documentation.

# Sources

- Fonts
  - `Tiny5 Regular` https://fonts.google.com/specimen/Tiny5?query=tiny5
  - `Kode Mono Regular` - https://fonts.google.com/specimen/Kode+Mono?query=kode+mono
- Assets
  - `16bit` - https://bz-game.itch.io/pixel-art-chess-set
  - `normal` - https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces