PYTHON PROJECT (2022-23)
EE-327

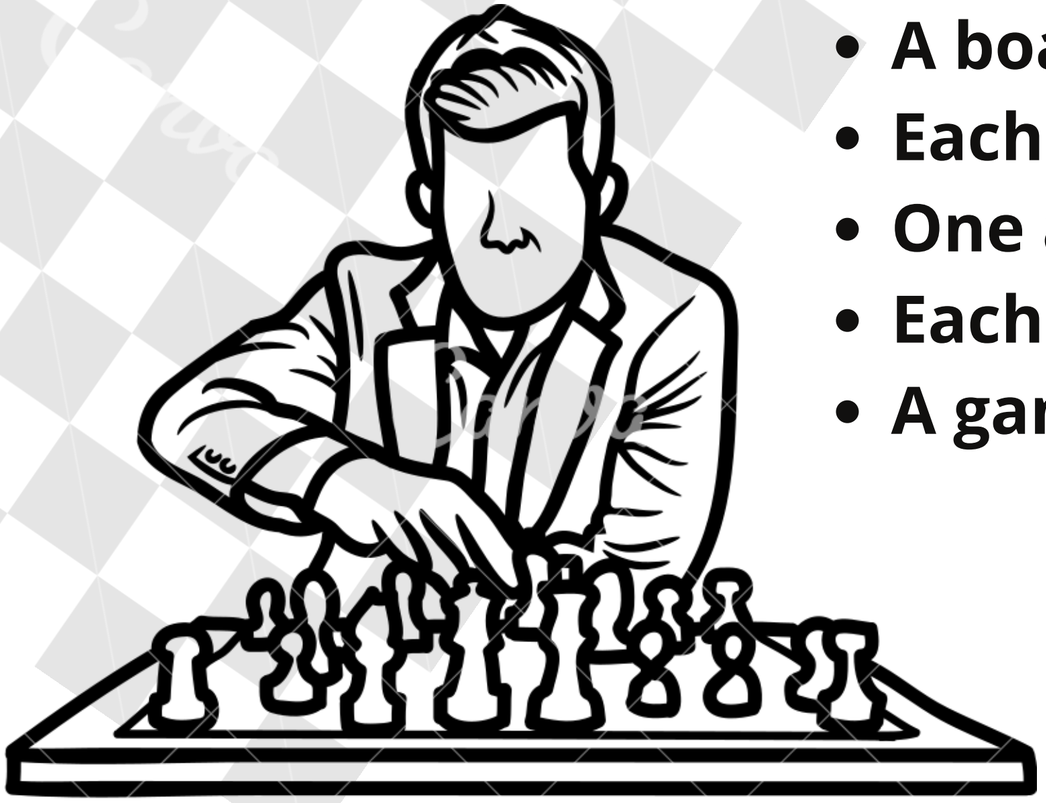
CHESS GAME

GAUTAM KUMAR 2K20/EE/105
HIMANSHU 2K20/EE/122
DISHU 2K20/EE/096

CONTENT

- Chess
- Scope of our project
- Building blocks of chess
- Tools and Techniques used
- Playing steps
- Implementation of program
- Structure of program
- Files
- Further scope

CHESS??

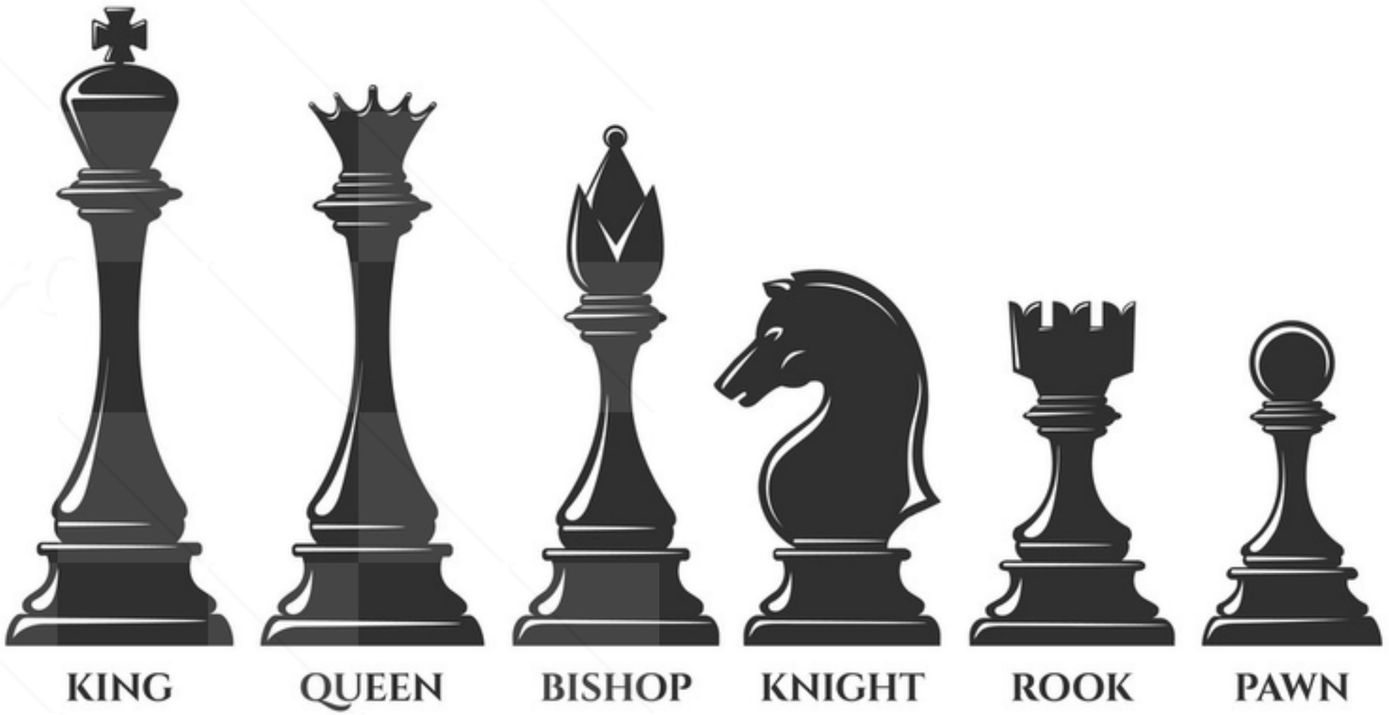


- A board game played by two people
- Each player control an army of 16 pieces
- One army is black, the other white
- Each piece moves in a unique way
- A game of strategy and concentration

OBJECTIVE

The goal of the game is to checkmate the opponent's king by placing it under an avoidable threat of capture.

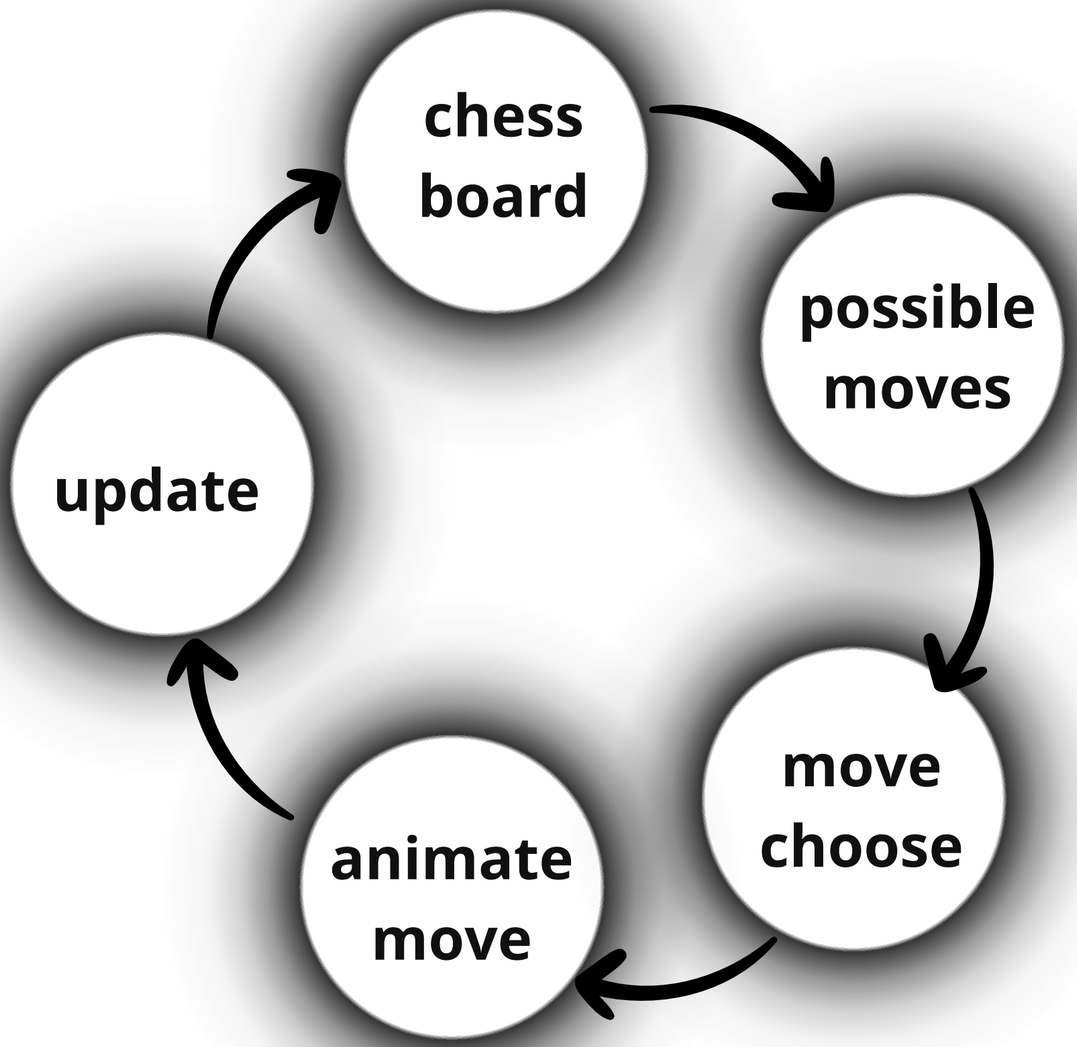
CHESS PIECES



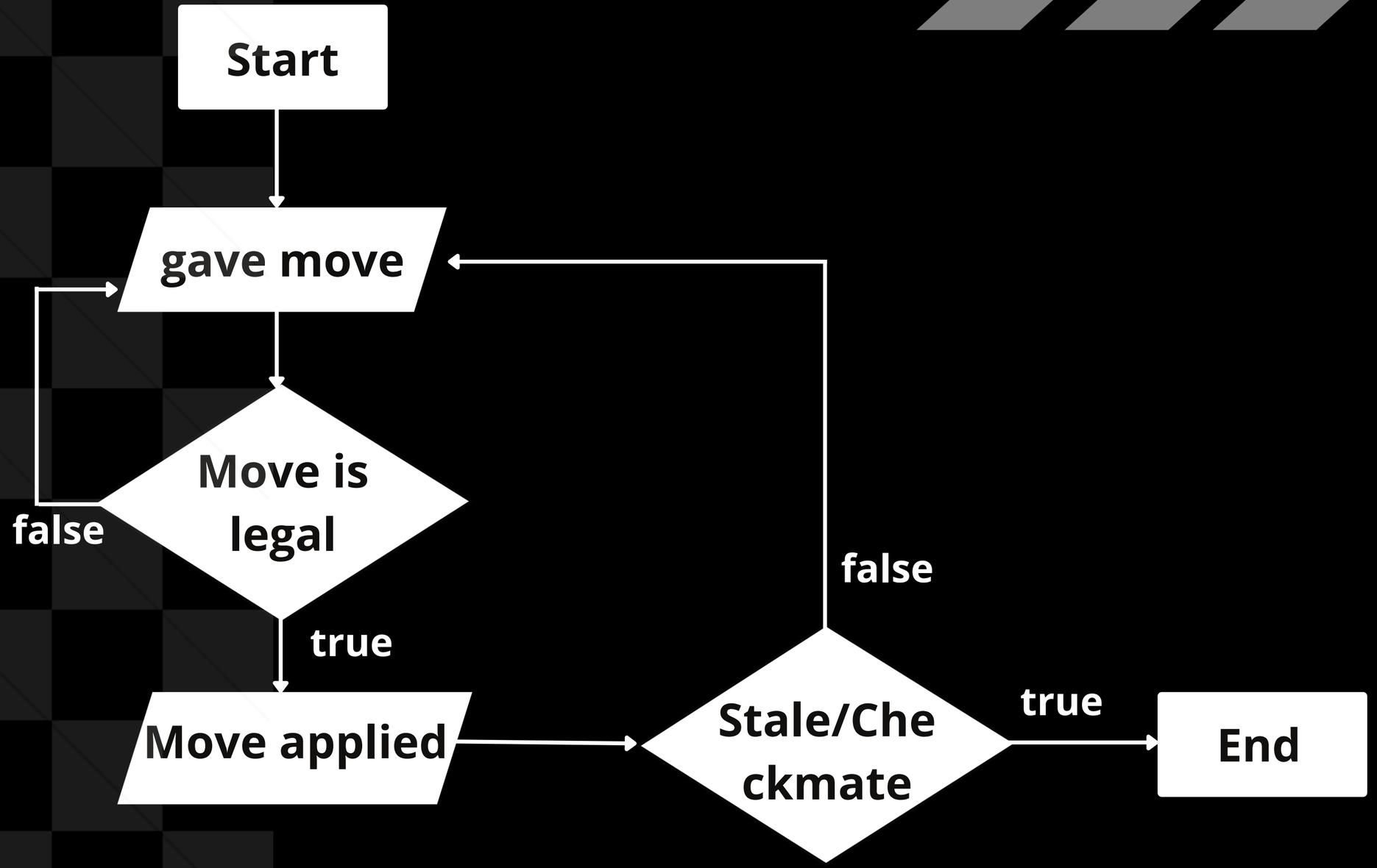
SCOPE OF OUR PROJECT

The scope of our project is limited to two players to play chess as real in computer.

BUILDING BLOCKS OF CHESS



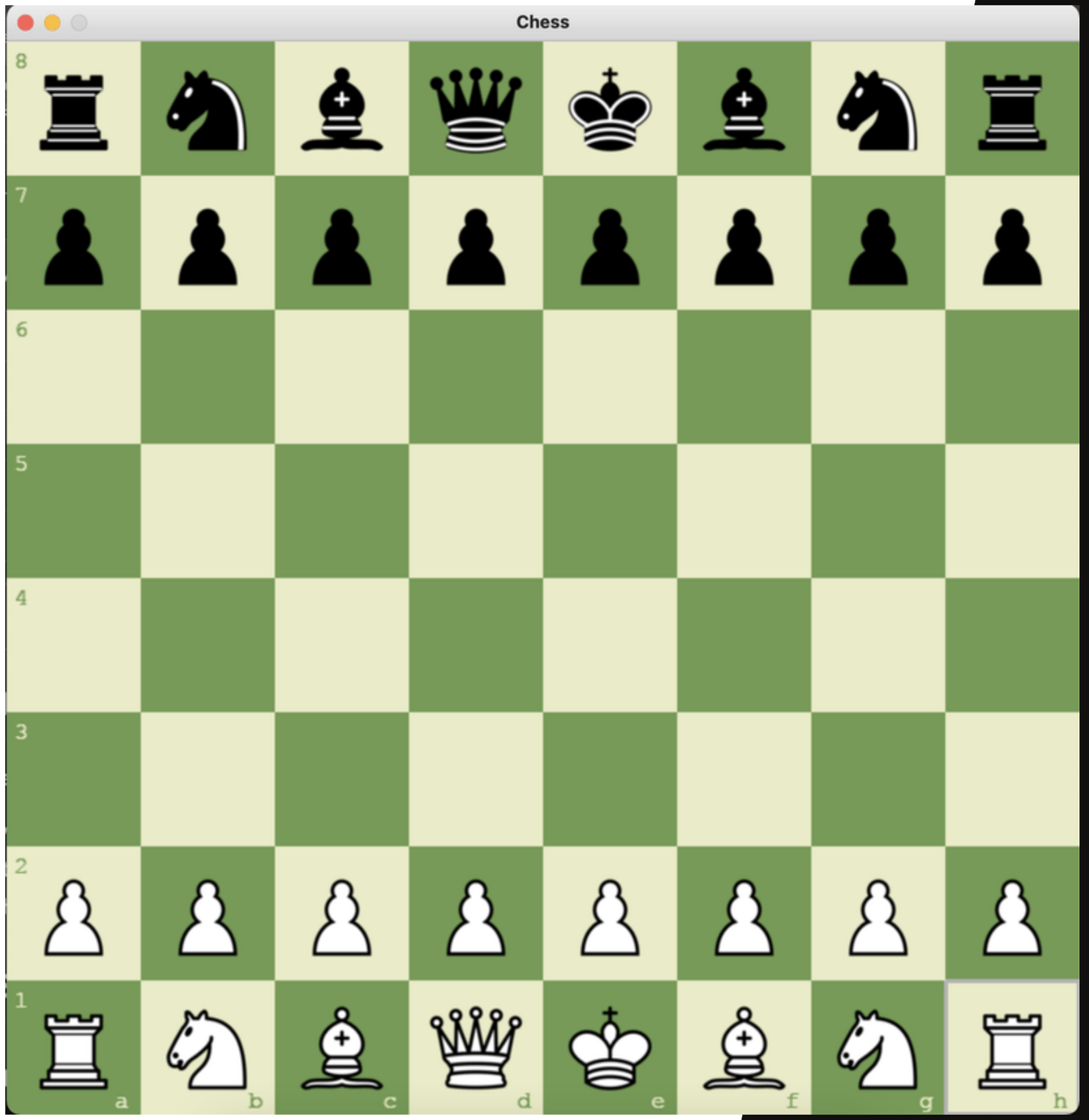
PLAYING STEPS



TOOLS AND TECHNIQUE USED

Python 3 : It is a General-purpose dynamic programming language, which provides the high-level readability and it is interpreted. In our project we use python to calculate the player's move.

Pygame : Pygame is a Python framework for game programming. we use pygame in our project for creating, updating and handling GUI.


Chess
8
7
6
5
4
3
2
1
a
b
c
d
e
f
g
h

IMPLEMENTATION OF PROGRAM

FILES

- **board.py -** used to create and design grid/ board of the game.
- **color.py -** responsible for colors
- **config.py -** responsible for the configuration(font, theme etc)
- **const.py -** save main contants that are needed for the game
- **dragger.py -** responsible for letting us drag any of these pieces all around the grid.
- **game.py -** responsible for all rendering methods
- **main.py -** have attributes - screen and method - main loop
- **move.py -** responsible for saving a move
- **piece.py -** to create the pieces
- **square.py -** part of grid that contains pieces
- **theme.py -** responsible for colors of trace, background, moves
- **sound.py -** it has two references move sound and capture sound

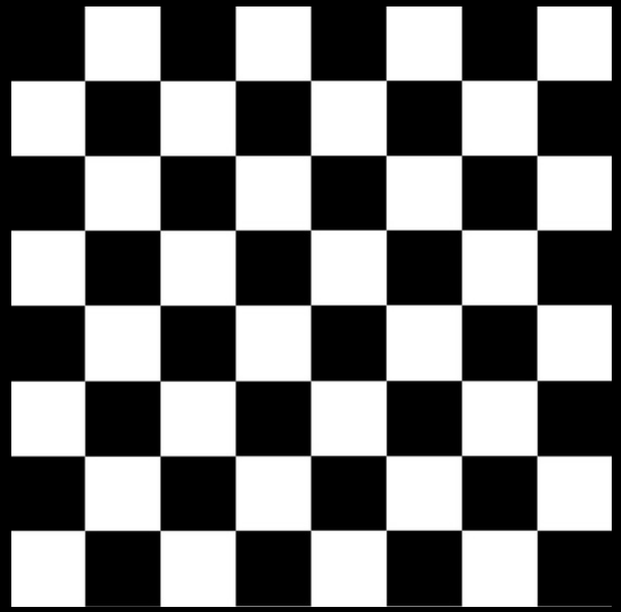
FURTHER SCOPE

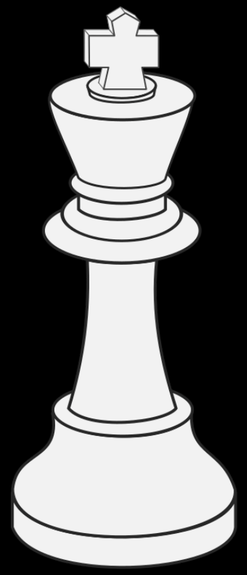
we can make it player vs computer (intelligent chess engine)

we can introduce two things to computer for making the game intelligent which will make the game to do optimal moves

- A technique to choose the move to make amongst all legal possibilities, so that it can choose a move instead of being forced to pick one at random.
- A way to compare moves and positions, so that it make intelligent choices.

Here we need to use some Deep Learning and AI tools like TensorFlow.
One famous algorithm Minimax can be used to estimate and evaluate the move
Techniques like aplha prunning search and can also be explored and tried to implement in the project.





MINIMAX ALGO

```python
def minimax(board, depth, maximizing_player):
    if depth == 0 or board.is_game_over():
        return evaluate(board)
    if maximizing_player:
        value = -float('inf')
        for move in board.legal_moves:
            board.push(move)
            value = max(value, minimax(board, depth - 1, Fal
            board.pop()
        return value
    else:
        value = float('inf')
        for move in board.legal_moves:
            board.push(move)
            value = min(value, minimax(board, depth - 1, Tru
            board.pop()
        return value
```

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Evaluation Technique

```python
import chess
piece_values = {
    chess.PAWN: 100,
    chess.ROOK: 500,
    chess.KNIGHT: 320,
    chess.BISHOP: 330,
    chess.QUEEN: 900,
    chess.KING: 20000
}
board = chess.Board(chess.STARTING_FEN)
white_material = 0
black_material = 0

for square in chess.SQUARES:
    piece = board.piece_at(square)
    if not piece:
        continue
    if piece.color == chess.WHITE:
        white_material += piece_values[piece.piece_type]
    else:
        black_material += piece_values[piece.piece_type]
```

It can be as simple as an indicator for whether a given side has been checkmated, but for limited-depth search this isn’t useful. A slightly more mature estimate is in counting the pieces on each side in a weighed way. If white has no queen but black does, then the position is unbalanced and white is in trouble. If white is three pawns down but also has an extra bishop, the position is likely to be balanced. This evaluation function scores pieces in terms of how many pawns they’re equivalent to.

THANK YOU

board.py

```
from const import *
from square import Square
from piece import *
from move import Move
from sound import Sound
import copy
import os

class Board:

    def __init__(self):
        self.squares = [[0, 0, 0, 0, 0, 0, 0, 0] for col in range(COLS)]
        self.last_move = None
        self._create()
        self._add_pieces('white')
        self._add_pieces('black')

    def move(self, piece, move, testing=False):
        initial = move.initial
        final = move.final

        en_passant_empty = self.squares[final.row][final.col].isempty()

        # console board move update
        self.squares[initial.row][initial.col].piece = None
        self.squares[final.row][final.col].piece = piece

        if isinstance(piece, Pawn):
            # en passant capture
            diff = final.col - initial.col
            if diff != 0 and en_passant_empty:
                # console board move update
                self.squares[initial.row][initial.col + diff].piece = None
                self.squares[final.row][final.col].piece = piece
                if not testing:
                    sound = Sound(
                        os.path.join('assets/sounds/capture.wav'))
                    sound.play()

            # pawn promotion
            else:
                self.check_promotion(piece, final)

        # king castling
        if isinstance(piece, King):
            if self.castling(initial, final) and not testing:
                diff = final.col - initial.col
                rook = piece.left_rook if (diff < 0) else piece.right_rook
                self.move(rook, rook.moves[-1])

        # move
        piece.moved = True

        # clear valid moves
        piece.clear_moves()

        # set last move
        self.last_move = move
```

```
def valid_move(self, piece, move):
    return move in piece.moves

def check_promotion(self, piece, final):
    if final.row == 0 or final.row == 7:
        self.squares[final.row][final.col].piece = Queen(piece.color)

def castling(self, initial, final):
    return abs(initial.col - final.col) == 2

def set_true_en_passant(self, piece):

    if not isinstance(piece, Pawn):
        return

    for row in range(ROWS):
        for col in range(COLS):
            if isinstance(self.squares[row][col].piece, Pawn):
                self.squares[row][col].piece.en_passant = False

    piece.en_passant = True

def in_check(self, piece, move):
    temp_piece = copy.deepcopy(piece)
    temp_board = copy.deepcopy(self)
    temp_board.move(temp_piece, move, testing=True)

    for row in range(ROWS):
        for col in range(COLS):
            if temp_board.squares[row][col].has_enemy_piece(piece.color):
                p = temp_board.squares[row][col].piece
                temp_board.calc_moves(p, row, col, bool=False)
                for m in p.moves:
                    if isinstance(m.final.piece, King):
                        return True

    return False

def calc_moves(self, piece, row, col, bool=True):
    '''
        Calculate all the possible (valid) moves of an specific piece on a specific position
    '''

    def pawn_moves():
        # steps
        steps = 1 if piece.moved else 2

        # vertical moves
        start = row + piece.dir
        end = row + (piece.dir * (1 + steps))
        for possible_move_row in range(start, end, piece.dir):
            if Square.in_range(possible_move_row):
                if self.squares[possible_move_row][col].isempty():
                    # create initial and final move squares
                    initial = Square(row, col)
                    final = Square(possible_move_row, col)
                    # create a new move
                    move = Move(initial, final)
```

```
                    # check potencial checks
                    if bool:
                        if not self.in_check(piece, move):
                            # append new move
                            piece.add_move(move)
                    else:
                        # append new move
                        piece.add_move(move)
                # blocked
                else: break
            # not in range
            else: break

        # diagonal moves
        possible_move_row = row + piece.dir
        possible_move_cols = [col-1, col+1]
        for possible_move_col in possible_move_cols:
            if Square.in_range(possible_move_row, possible_move_col):
                if self.squares[possible_move_row]
[possible_move_col].has_enemy_piece(piece.color):
                    # create initial and final move squares
                    initial = Square(row, col)
                    final_piece = self.squares[possible_move_row][possible_move_col].piece
                    final = Square(possible_move_row, possible_move_col, final_piece)
                    # create a new move
                    move = Move(initial, final)

                    # check potencial checks
                    if bool:
                        if not self.in_check(piece, move):
                            # append new move
                            piece.add_move(move)
                    else:
                        # append new move
                        piece.add_move(move)

        # en passant moves
        r = 3 if piece.color == 'white' else 4
        fr = 2 if piece.color == 'white' else 5
        # left en pessant
        if Square.in_range(col-1) and row == r:
            if self.squares[row][col-1].has_enemy_piece(piece.color):
                p = self.squares[row][col-1].piece
                if isinstance(p, Pawn):
                    if p.en_passant:
                        # create initial and final move squares
                        initial = Square(row, col)
                        final = Square(fr, col-1, p)
                        # create a new move
                        move = Move(initial, final)

                        # check potencial checks
                        if bool:
                            if not self.in_check(piece, move):
                                # append new move
                                piece.add_move(move)
                        else:
                            # append new move
                            piece.add_move(move)
```

```
            # right en pessant
            if Square.in_range(col+1) and row == r:
                if self.squares[row][col+1].has_enemy_piece(piece.color):
                    p = self.squares[row][col+1].piece
                    if isinstance(p, Pawn):
                        if p.en_passant:
                            # create initial and final move squares
                            initial = Square(row, col)
                            final = Square(fr, col+1, p)
                            # create a new move
                            move = Move(initial, final)

                            # check potencial checks
                            if bool:
                                if not self.in_check(piece, move):
                                    # append new move
                                    piece.add_move(move)
                            else:
                                # append new move
                                piece.add_move(move)


        def knight_moves():
            # 8 possible moves
            possible_moves = [
                (row-2, col+1),
                (row-1, col+2),
                (row+1, col+2),
                (row+2, col+1),
                (row+2, col-1),
                (row+1, col-2),
                (row-1, col-2),
                (row-2, col-1),
            ]

            for possible_move in possible_moves:
                possible_move_row, possible_move_col = possible_move

                if Square.in_range(possible_move_row, possible_move_col):
                    if self.squares[possible_move_row]
[possible_move_col].isempty_or_enemy(piece.color):
                        # create squares of the new move
                        initial = Square(row, col)
                        final_piece = self.squares[possible_move_row][possible_move_col].piece
                        final = Square(possible_move_row, possible_move_col, final_piece)
                        # create new move
                        move = Move(initial, final)

                        # check potencial checks
                        if bool:
                            if not self.in_check(piece, move):
                                # append new move
                                piece.add_move(move)
                            else: break
                        else:
                            # append new move
                            piece.add_move(move)

        def straightline_moves(incrs):
            for incr in incrs:
```

```
            row_incr, col_incr = incr
            possible_move_row = row + row_incr
            possible_move_col = col + col_incr

            while True:
                if Square.in_range(possible_move_row, possible_move_col):
                    # create squares of the possible new move
                    initial = Square(row, col)
                    final_piece = self.squares[possible_move_row][possible_move_col].piece
                    final = Square(possible_move_row, possible_move_col, final_piece)
                    # create a possible new move
                    move = Move(initial, final)

                    # empty = continue looping
                    if self.squares[possible_move_row][possible_move_col].isempty():
                        # check potencial checks
                        if bool:
                            if not self.in_check(piece, move):
                                # append new move
                                piece.add_move(move)
                        else:
                            # append new move
                            piece.add_move(move)

                    # has enemy piece = add move + break
                    elif self.squares[possible_move_row]
[possible_move_col].has_enemy_piece(piece.color):
                        # check potencial checks
                        if bool:
                            if not self.in_check(piece, move):
                                # append new move
                                piece.add_move(move)
                        else:
                            # append new move
                            piece.add_move(move)
                        break

                    # has team piece = break
                    elif self.squares[possible_move_row]
[possible_move_col].has_team_piece(piece.color):
                        break

                # not in range
                else: break

                # incrementing incrs
                possible_move_row = possible_move_row + row_incr
                possible_move_col = possible_move_col + col_incr

        def king_moves():
            adjs = [
                (row-1, col+0), # up
                (row-1, col+1), # up-right
                (row+0, col+1), # right
                (row+1, col+1), # down-right
                (row+1, col+0), # down
                (row+1, col-1), # down-left
                (row+0, col-1), # left
                (row-1, col-1), # up-left
            ]
```

```
        # normal moves
        for possible_move in adjs:
            possible_move_row, possible_move_col = possible_move

            if Square.in_range(possible_move_row, possible_move_col):
                if self.squares[possible_move_row]
[possible_move_col].isempty_or_enemy(piece.color):
                    # create squares of the new move
                    initial = Square(row, col)
                    final = Square(possible_move_row, possible_move_col) # piece=piece
                    # create new move
                    move = Move(initial, final)
                    # check potencial checks
                    if bool:
                        if not self.in_check(piece, move):
                            # append new move
                            piece.add_move(move)
                        else: break
                    else:
                        # append new move
                        piece.add_move(move)

        # castling moves
        if not piece.moved:
            # queen castling
            left_rook = self.squares[row][0].piece
            if isinstance(left_rook, Rook):
                if not left_rook.moved:
                    for c in range(1, 4):
                        # castling is not possible because there are pieces in between ?
                        if self.squares[row][c].has_piece():
                            break

                        if c == 3:
                            # adds left rook to king
                            piece.left_rook = left_rook

                            # rook move
                            initial = Square(row, 0)
                            final = Square(row, 3)
                            moveR = Move(initial, final)

                            # king move
                            initial = Square(row, col)
                            final = Square(row, 2)
                            moveK = Move(initial, final)

                            # check potencial checks
                            if bool:
                                if not self.in_check(piece, moveK) and not self.in_check(left_rook, moveR):
                                    # append new move to rook
                                    left_rook.add_move(moveR)
                                    # append new move to king
                                    piece.add_move(moveK)
                            else:
                                # append new move to rook
                                left_rook.add_move(moveR)
                                # append new move king
                                piece.add_move(moveK)
```

```
            # king castling
            right_rook = self.squares[row][7].piece
            if isinstance(right_rook, Rook):
                if not right_rook.moved:
                    for c in range(5, 7):
                        # castling is not possible because there are pieces in between ?
                        if self.squares[row][c].has_piece():
                            break

                        if c == 6:
                            # adds right rook to king
                            piece.right_rook = right_rook

                            # rook move
                            initial = Square(row, 7)
                            final = Square(row, 5)
                            moveR = Move(initial, final)

                            # king move
                            initial = Square(row, col)
                            final = Square(row, 6)
                            moveK = Move(initial, final)

                            # check potencial checks
                            if bool:
                                if not self.in_check(piece, moveK) and not self.in_check(right_rook, moveR):
                                    # append new move to rook
                                    right_rook.add_move(moveR)
                                    # append new move to king
                                    piece.add_move(moveK)
                            else:
                                # append new move to rook
                                right_rook.add_move(moveR)
                                # append new move king
                                piece.add_move(moveK)

        if isinstance(piece, Pawn):
            pawn_moves()

        elif isinstance(piece, Knight):
            knight_moves()

        elif isinstance(piece, Bishop):
            straightline_moves([
                (-1, 1), # up-right
                (-1, -1), # up-left
                (1, 1), # down-right
                (1, -1), # down-left
            ])

        elif isinstance(piece, Rook):
            straightline_moves([
                (-1, 0), # up
                (0, 1), # right
                (1, 0), # down
                (0, -1), # left
            ])

        elif isinstance(piece, Queen):
```

```
            straightline_moves([
                (-1, 1), # up-right
                (-1, -1), # up-left
                (1, 1), # down-right
                (1, -1), # down-left
                (-1, 0), # up
                (0, 1), # right
                (1, 0), # down
                (0, -1) # left
            ])

        elif isinstance(piece, King):
            king_moves()

    def _create(self):
        for row in range(ROWS):
            for col in range(COLS):
                self.squares[row][col] = Square(row, col)

    def _add_pieces(self, color):
        row_pawn, row_other = (6, 7) if color == 'white' else (1, 0)

        # pawns
        for col in range(COLS):
            self.squares[row_pawn][col] = Square(row_pawn, col, Pawn(color))

        # knights
        self.squares[row_other][1] = Square(row_other, 1, Knight(color))
        self.squares[row_other][6] = Square(row_other, 6, Knight(color))

        # bishops
        self.squares[row_other][2] = Square(row_other, 2, Bishop(color))
        self.squares[row_other][5] = Square(row_other, 5, Bishop(color))

        # rooks
        self.squares[row_other][0] = Square(row_other, 0, Rook(color))
        self.squares[row_other][7] = Square(row_other, 7, Rook(color))

        # queen
        self.squares[row_other][3] = Square(row_other, 3, Queen(color))

        # king
        self.squares[row_other][4] = Square(row_other, 4, King(color))
```

Color.py

```
class Color:

    def __init__(self, light, dark):
        self.light = light
        self.dark = dark
```

Config.py

```
import pygame
import os

from sound import Sound
from theme import Theme

class Config:

    def __init__(self):
        self.themes = []
        self._add_themes()
        self.idx = 0
        self.theme = self.themes[self.idx]
        self.font = pygame.font.SysFont('monospace', 18, bold=True)
        self.move_sound = Sound(
            os.path.join('assets/sounds/move.wav'))
        self.capture_sound = Sound(
            os.path.join('assets/sounds/capture.wav'))

    def change_theme(self):
        self.idx += 1
        self.idx %= len(self.themes)
        self.theme = self.themes[self.idx]

    def _add_themes(self):
        green = Theme((234, 235, 200), (119, 154, 88), (244, 247, 116), (172, 195, 51), '#C86464',
'#C84646')
        brown = Theme((235, 209, 166), (165, 117, 80), (245, 234, 100), (209, 185, 59), '#C86464',
'#C84646')
        blue = Theme((229, 228, 200), (60, 95, 135), (123, 187, 227), (43, 119, 191), '#C86464',
'#C84646')
        gray = Theme((120, 119, 118), (86, 85, 84), (99, 126, 143), (82, 102, 128), '#C86464',
'#C84646')

        self.themes = [green, brown, blue, gray]
```

Constant.py

```
# Screen dimensions
WIDTH = 800
HEIGHT = 800

# Board dimensions
ROWS = 8
COLS = 8
SQSIZE = WIDTH // COLS
```

Dragger.py

```
import pygame

from const import *

class Dragger:

    def __init__(self):
        self.piece = None
        self.dragging = False
        self.mouseX = 0
        self.mouseY = 0
        self.initial_row = 0
        self.initial_col = 0

    # blit method

    def update_blit(self, surface):
        # texture
        self.piece.set_texture(size=128)
        texture = self.piece.texture
        # img
        img = pygame.image.load(texture)
        # rect
        img_center = (self.mouseX, self.mouseY)
        self.piece.texture_rect = img.get_rect(center=img_center)
        # blit
        surface.blit(img, self.piece.texture_rect)

    # other methods

    def update_mouse(self, pos):
        self.mouseX, self.mouseY = pos # (xcor, ycor)

    def save_initial(self, pos):
        self.initial_row = pos[1] // SQSIZE
        self.initial_col = pos[0] // SQSIZE

    def drag_piece(self, piece):
        self.piece = piece
        self.dragging = True

    def undrag_piece(self):
        self.piece = None
        self.dragging = False
```

Game.py

```
import pygame

from const import *
from board import Board
from dragger import Dragger
from config import Config
from square import Square

class Game:

    def __init__(self):
        self.next_player = 'white'
        self.hovered_sqr = None
        self.board = Board()
        self.dragger = Dragger()
        self.config = Config()

    # blit methods

    def show_bg(self, surface):
        theme = self.config.theme

        for row in range(ROWS):
            for col in range(COLS):
                # color
                color = theme.bg.light if (row + col) % 2 == 0 else theme.bg.dark
                # rect
                rect = (col * SQSIZE, row * SQSIZE, SQSIZE, SQSIZE)
                # blit
                pygame.draw.rect(surface, color, rect)

                # row coordinates
                if col == 0:
                    # color
                    color = theme.bg.dark if row % 2 == 0 else theme.bg.light
                    # label
                    lbl = self.config.font.render(str(ROWS-row), 1, color)
                    lbl_pos = (5, 5 + row * SQSIZE)
                    # blit
                    surface.blit(lbl, lbl_pos)

                # col coordinates
                if row == 7:
                    # color
                    color = theme.bg.dark if (row + col) % 2 == 0 else theme.bg.light
                    # label
                    lbl = self.config.font.render(Square.get_alphacol(col), 1, color)
                    lbl_pos = (col * SQSIZE + SQSIZE - 20, HEIGHT - 20)
                    # blit
                    surface.blit(lbl, lbl_pos)

    def show_pieces(self, surface):
        for row in range(ROWS):
            for col in range(COLS):
                # piece ?
                if self.board.squares[row][col].has_piece():
                    piece = self.board.squares[row][col].piece
```

```
                # all pieces except dragger piece
                if piece is not self.dragger.piece:
                    piece.set_texture(size=80)
                    img = pygame.image.load(piece.texture)
                    img_center = col * SQSIZE + SQSIZE // 2, row * SQSIZE + SQSIZE // 2
                    piece.texture_rect = img.get_rect(center=img_center)
                    surface.blit(img, piece.texture_rect)

    def show_moves(self, surface):
        theme = self.config.theme

        if self.dragger.dragging:
            piece = self.dragger.piece

            # loop all valid moves
            for move in piece.moves:
                # color
                color = theme.moves.light if (move.final.row + move.final.col) % 2 == 0 else
theme.moves.dark
                # rect
                rect = (move.final.col * SQSIZE, move.final.row * SQSIZE, SQSIZE, SQSIZE)
                # blit
                pygame.draw.rect(surface, color, rect)

    def show_last_move(self, surface):
        theme = self.config.theme

        if self.board.last_move:
            initial = self.board.last_move.initial
            final = self.board.last_move.final

            for pos in [initial, final]:
                # color
                color = theme.trace.light if (pos.row + pos.col) % 2 == 0 else theme.trace.dark
                # rect
                rect = (pos.col * SQSIZE, pos.row * SQSIZE, SQSIZE, SQSIZE)
                # blit
                pygame.draw.rect(surface, color, rect)

    def show_hover(self, surface):
        if self.hovered_sqr:
            # color
            color = (180, 180, 180)
            # rect
            rect = (self.hovered_sqr.col * SQSIZE, self.hovered_sqr.row * SQSIZE, SQSIZE, SQSIZE)
            # blit
            pygame.draw.rect(surface, color, rect, width=3)

    # other methods

    def next_turn(self):
        self.next_player = 'white' if self.next_player == 'black' else 'black'

    def set_hover(self, row, col):
        self.hovered_sqr = self.board.squares[row][col]

    def change_theme(self):
        self.config.change_theme()

    def play_sound(self, captured=False):
```

```
        if captured:
            self.config.capture_sound.play()
        else:
            self.config.move_sound.play()

    def reset(self):
        self.__init__()
```

Main.py

```
import pygame
import sys

from const import *
from game import Game
from square import Square
from move import Move

class Main:

    def __init__(self):
        pygame.init()
        self.screen = pygame.display.set_mode( (WIDTH, HEIGHT) )
        pygame.display.set_caption('Chess')
        self.game = Game()

    def mainloop(self):

        screen = self.screen
        game = self.game
        board = self.game.board
        dragger = self.game.dragger

        while True:
            # show methods
            game.show_bg(screen)
            game.show_last_move(screen)
            game.show_moves(screen)
            game.show_pieces(screen)
            game.show_hover(screen)

            if dragger.dragging:
                dragger.update_blit(screen)

            for event in pygame.event.get():

                # click
                if event.type == pygame.MOUSEBUTTONDOWN:
                    dragger.update_mouse(event.pos)

                    clicked_row = dragger.mouseY // SQSIZE
                    clicked_col = dragger.mouseX // SQSIZE

                    # if clicked square has a piece ?
                    if board.squares[clicked_row][clicked_col].has_piece():
                        piece = board.squares[clicked_row][clicked_col].piece
                        # valid piece (color) ?
                        if piece.color == game.next_player:
                            board.calc_moves(piece, clicked_row, clicked_col, bool=True)
```

```
                dragger.save_initial(event.pos)
                dragger.drag_piece(piece)
                # show methods
                game.show_bg(screen)
                game.show_last_move(screen)
                game.show_moves(screen)
                game.show_pieces(screen)

        # mouse motion
        elif event.type == pygame.MOUSEMOTION:
            motion_row = event.pos[1] // SQSIZE
            motion_col = event.pos[0] // SQSIZE

            game.set_hover(motion_row, motion_col)

            if dragger.dragging:
                dragger.update_mouse(event.pos)
                # show methods
                game.show_bg(screen)
                game.show_last_move(screen)
                game.show_moves(screen)
                game.show_pieces(screen)
                game.show_hover(screen)
                dragger.update_blit(screen)

        # click release
        elif event.type == pygame.MOUSEBUTTONUP:

            if dragger.dragging:
                dragger.update_mouse(event.pos)

                released_row = dragger.mouseY // SQSIZE
                released_col = dragger.mouseX // SQSIZE

                # create possible move
                initial = Square(dragger.initial_row, dragger.initial_col)
                final = Square(released_row, released_col)
                move = Move(initial, final)

                # valid move ?
                if board.valid_move(dragger.piece, move):
                    # normal capture
                    captured = board.squares[released_row][released_col].has_piece()
                    board.move(dragger.piece, move)

                    board.set_true_en_passant(dragger.piece)

                    # sounds
                    game.play_sound(captured)
                    # show methods
                    game.show_bg(screen)
                    game.show_last_move(screen)
                    game.show_pieces(screen)
                    # next turn
                    game.next_turn()

            dragger.undrag_piece()

        # key press
        elif event.type == pygame.KEYDOWN:
```

```
                # changing themes
                if event.key == pygame.K_t:
                    game.change_theme()

                 # changing themes
                if event.key == pygame.K_r:
                    game.reset()
                    game = self.game
                    board = self.game.board
                    dragger = self.game.dragger

            # quit application
            elif event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()

        pygame.display.update()


main = Main()
main.mainloop()
```

move.py

```
class Move:

    def __init__(self, initial, final):
        # initial and final are squares
        self.initial = initial
        self.final = final

    def __str__(self):
        s = ''
        s += f'({self.initial.col}, {self.initial.row})'
        s += f' -> ({self.final.col}, {self.final.row})'
        return s

    def __eq__(self, other):
        return self.initial == other.initial and self.final == other.final
```

Piece.py

```
import os

class Piece:

    def __init__(self, name, color, value, texture=None, texture_rect=None):
        self.name = name
        self.color = color
        value_sign = 1 if color == 'white' else -1
        self.value = value * value_sign
        self.moves = []
        self.moved = False
        self.texture = texture
        self.set_texture()
```

```
        self.texture_rect = texture_rect

    def set_texture(self, size=80):
        self.texture = os.path.join(
            f'assets/images/imgs-{size}px/{self.color}_{self.name}.png')

    def add_move(self, move):
        self.moves.append(move)

    def clear_moves(self):
        self.moves = []

class Pawn(Piece):

    def __init__(self, color):
        self.dir = -1 if color == 'white' else 1
        self.en_passant = False
        super().__init__('pawn', color, 1.0)

class Knight(Piece):

    def __init__(self, color):
        super().__init__('knight', color, 3.0)

class Bishop(Piece):

    def __init__(self, color):
        super().__init__('bishop', color, 3.001)

class Rook(Piece):

    def __init__(self, color):
        super().__init__('rook', color, 5.0)

class Queen(Piece):

    def __init__(self, color):
        super().__init__('queen', color, 9.0)

class King(Piece):

    def __init__(self, color):
        self.left_rook = None
        self.right_rook = None
        super().__init__('king', color, 10000.0)
```

Sound.py

```
import pygame

class Sound:

    def __init__(self, path):
        self.path = path
        self.sound = pygame.mixer.Sound(path)

    def play(self):
        pygame.mixer.Sound.play(self.sound)
```

Square.py

```
class Square:

    ALPHACOLS = {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h'}

    def __init__(self, row, col, piece=None):
        self.row = row
        self.col = col
        self.piece = piece
        self.alphacol = self.ALPHACOLS[col]

    def __eq__(self, other):
        return self.row == other.row and self.col == other.col
    def has_piece(self):
        return self.piece != None

    def isempty(self):
        return not self.has_piece()

    def has_team_piece(self, color):
        return self.has_piece() and self.piece.color == color

    def has_enemy_piece(self, color):
        return self.has_piece() and self.piece.color != color

    def isempty_or_enemy(self, color):
        return self.isempty() or self.has_enemy_piece(color)

    @staticmethod
    def in_range(*args):
        for arg in args:
            if arg < 0 or arg > 7:
                return False

        return True

    @staticmethod
    def get_alphacol(col):
        ALPHACOLS = {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h'}
        return ALPHACOLS[col]
```

Theme.py

```
from color import Color
#here we have used color moudule because
#Converts and manipulates common color representation (RGB, HSL, web, …)
class Theme:

    def __init__(self, light_bg, dark_bg,
                       light_trace, dark_trace,
                       light_moves, dark_moves):

        self.bg = Color(light_bg, dark_bg)
        self.trace = Color(light_trace, dark_trace)
        self.moves = Color(light_moves, dark_moves)
```