Introduction to Programming
Semester 1/2021

Python Project Report

CheckM8 Chess

Progress log & development timestamp

Made & Developed by:

Mr. Jirapat Wongjaroenrat 64011405
Double-Degree Program in Financial Engineering (International Program)

Presented to:
Dr. Natthapong Jungteerapanich (Instructor)

https://www.chessprogramming.org/

20/10/2021

Creating Chess Engine UI with pygame

sEngine.py × ChessMain.py ×

```python
"""
This is our main driver file. It will be responsible for handling user input
and displaying the current Game state
"""

import pygame as p
from Chess import ChessEngine

WIDTH = HEIGHT = 512  # 400 another option
DIMENSION = 8  # dimensions of a chess board are 8x8
SQ_SIZE = HEIGHT // DIMENSION
MAX_FPS = 15  # for animations
IMAGES = {}

# loading Images .png, own method instead of putting in main. To support picking
# couple different chess sets options later on.
"""
Initialize a global dictionary of images. This will be called exactly once in the main
"""


def loadImages():
    white_chess_pieces = ["wp", "wR", "wN", "wB", "wQ", "wK"]
    for piece in white_chess_pieces:
        IMAGES[piece] = p.image.load("images/" + piece + ".png")
    # Note: access an image by using 'IMAGES['wp']', return wp image. Check draw image not
    # Change image directory in 'images/'

    black_chess_pieces = ["bp", "bR", "bN", "bB", "bQ", "bK"]
    for piece in black_chess_pieces:
        IMAGES[piece] = p.image.load("images/" + piece + ".png")  # p.transform.scale(p.ima
    # Note: access an image by using 'IMAGES['bp']'
    # Change image directory in 'images/'
```

ChessEngine.py × ChessMain.py ×

```python
"""
This class is responsible for storing all the information about the current state of
a chess game. It will also be responsible for determining the valid moves at the current state.
It kill also keep a chess-move log (undo move, check opponent & your current moves)
"""


class GameState():
    """Chess board"""  # 2 dimensional-list  8x8 matrix

    def __init__(self):  # each list represent a row on the chess board
        # board is an 8x8 2d list, each element of the list has 2 characters
        # The first character represents the color of the piece, 'b' or 'w'
        # The second character represents the type of the piece, 'K', 'Q', 'R', 'B', 'N' or 'p'
        # "--" represents an empty space on chess board with no piece.
        self.board = [
            ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],
            ["bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],
            ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"],
        ]
        self.whiteToMove = True
        self.moveLog = []
```

Testing import local Module from :ChessEngine: into ChessMain

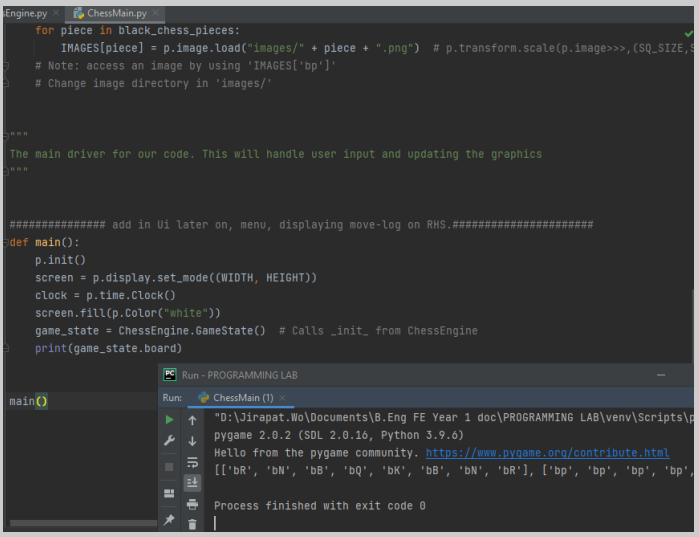
Engine.py | ChessMain.py

```
"""
This is our main driver file. It will be responsible for handling user input
and displaying the current Game state
"""

import pygame as p
from Chess import ChessEngine
```

20/10/2021

```
"""
The main driver for our code. This will handle user input and updating the graphics
"""


################ add in Ui later on, menu, displaying move-log on RHS.################
def main():
    p.init()
    screen = p.display.set_mode((WIDTH, HEIGHT))
    clock = p.time.Clock()
    screen.fill(p.Color("white"))
    game_state = ChessEngine.GameState()  # Calls _init_ from ChessEngine
    print(game_state.board)


main()
```

Calls __init__ which contains 2d list of 8x8 matrix (chess-board) from ChessEngine module

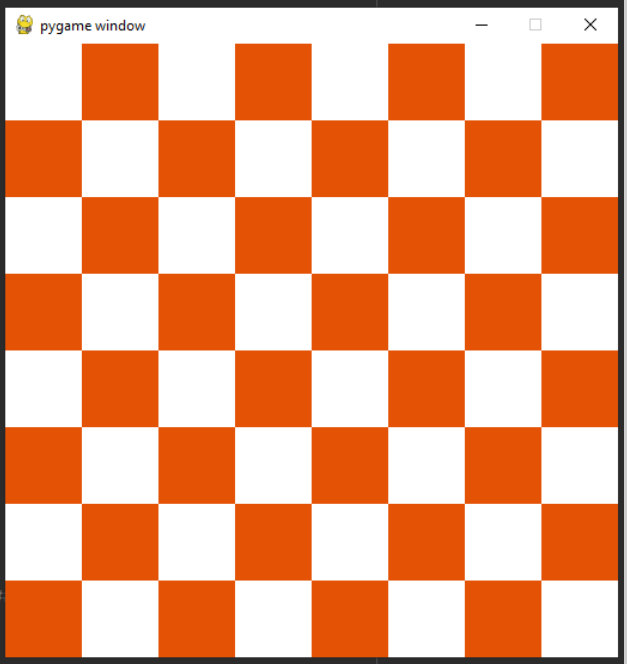


Test successfully.

```
"D:\Jirapat.Wo\Documents\B.Eng FE Year 1 doc\PROGRAMMING LAB\venv\Scripts\python.exe" "D:/Jirapat.Wo
pygame 2.0.2 (SDL 2.0.16, Python 3.9.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
[['bR', 'bN', 'bB', 'bQ', 'bK', 'bB', 'bN', 'bR'], ['bp', 'bp', 'bp', 'bp', 'bp', 'bp', 'bp', 'bp'],

Process finished with exit code 0
```

Making the chess board and basic gui

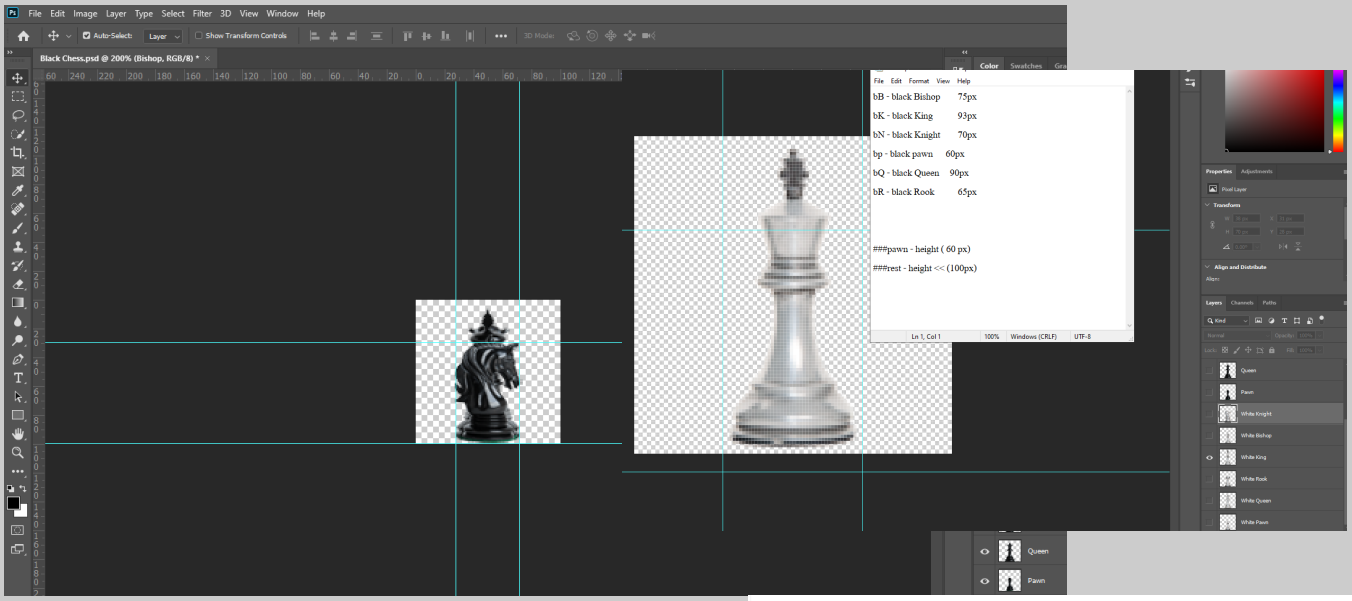
```
################ add in Ui later on, menu, displaying move-log on RHS.######################
def main():
    p.init()
    screen = p.display.set_mode((WIDTH, HEIGHT))
    clock = p.time.Clock()
    screen.fill(p.Color("white"))
    game_state = ChessEngine.GameState()  # Calls _init_ from ChessEngine
    loadImages()  # do this once, before while loop
    running = True
    while running:
        for e in p.event.get():
            if e.type == p.QUIT:
                running = False
        drawGameState(screen, game_state)
        clock.tick(MAX_FPS)
        p.display.flip()


"""
Responsible for all the graphics within a current game state
"""


################ add in chess piece highlighting, move suggestions later #########
def drawGameState(screen, game_state):
    drawBoard(screen)  # draw squares on the board
    drawPieces(screen, game_state.board)  # draw chess pieces on top of those squares
```



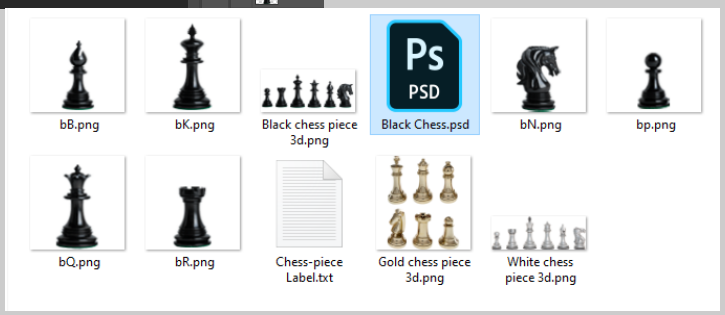

```
################ add in chess piece highlighting, move suggestions later ################
def drawGameState(screen, game_state):
    drawBoard(screen)  # draw squares on the board
    drawPieces(screen, game_state.board)  # draw chess pieces on top of those squares


"""
Draw the squares on the board. (Top Left square is always white)
"""


def drawBoard(screen):  # white (even), r = 2.  black (odd), r = 1
    colors = [p.Color("white"), p.Color("#E35205")]
    for r in range(DIMENSION):   # r = row , c = coluom
        for c in range(DIMENSION):
            color = colors[((r+c) % 2)]
            p.draw.rect(screen, color, p.Rect(c*SQ_SIZE, r*SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

21/10/2021

Load custom chess piece images on chess board



With the screen Width x Height is 800x800px, we can calculate each of the square sizes (SQ_SIZE) on the chess board by dividing the W, H by 8 (dimension). So each SQ_SIZE will be 100x100px. We can use the SQ_SIZE value to transform the image size to fit perfectly in each square with high resolution.




```
WIDTH = HEIGHT = 800  # 512, 400 another option
DIMENSION = 8  # dimensions of a chess board are 8x8
SQ_SIZE = HEIGHT // DIMENSION  # 100 x 100px per SQ , Image size <= 100, Canvas size <= 100
MAX_FPS = 15  # for animations
IMAGES = {}
```

```
# loading Images .png, own method instead of putting in main. To support picking
# couple different chess sets options later on.
"""
Initialize a global dictionary of images. This will be called exactly once in the main
"""


def loadImages():
    white_chess_pieces = ["wp", "wR", "wN", "wB", "wQ", "wK"]
    for piece in white_chess_pieces:
        IMAGES[piece] = p.transform.scale(p.image.load("images/" + piece + ".png"), (SQ_SIZE, SQ_SIZE))
    # Note: access an image by using 'IMAGES['wp']', return wp image. Check draw image not blank
    # Change image directory in 'images/'

    black_chess_pieces = ["bp", "bR", "bN", "bB", "bQ", "bK"]
    for piece in black_chess_pieces:
        IMAGES[piece] = p.transform.scale(p.image.load("Custom pieces/" + piece + ".png"),
                                          (SQ_SIZE, SQ_SIZE))  # p.transform.scale(p.image>>>,(SQ_SIZE,SQ_SIZE)
    # Note: access an image by using 'IMAGES['bp']'
    # Change image directory in 'images/'
```

Chess Main: Drawing Chess Pieces on ui

```python
def drawBoard(screen):  # white (even), r = 2.  black (odd), r = 1
    colors = [p.Color("white"), p.Color("#E35205")]  # KMITL Hex color
    for r in range(DIMENSION):  # r = row , c = coluom
        for c in range(DIMENSION):
            color = colors[((r + c) % 2)]
            p.draw.rect(screen, color, p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))


"""
Draw the pieces on the board using the current GameState.board
"""


def drawPieces(screen, board):
    for r in range(DIMENSION):
        for c in range(DIMENSION):
            piece = board[r][c]
            if piece != "--":  # not empty square
                screen.blit(IMAGES[piece], p.Rect(c * SQ_SIZE, r * SQ_SIZE, SQ_SIZE, SQ_SIZE))
```

Before

After

bB - black Bishop	75px
bK - black King	93px
bN - black Knight	70px
bp - black pawn	60px
bQ - black Queen	90px
bR - black Rook	65px

###pawn - height (60 px)

###rest - height << (100px)

22/10/21

Chess Main: Making Chess Moves

```
############## Game loop : Don't touch ##############
running = True
sqSelected = ()  # no square is selected, keep track of the last click of the user (tuple: (row, col))
playerClicks = []  # keep track of player clicks (two tuples: [(6, 4), (4, 4)])
while running:
    for event in p.event.get():
        if event.type == p.USEREVENT:  # A track has ended
            if len(playlist) > 0:  # If there are more tracks in the queue...
                p.mixer.music.queue(playlist.pop())  # Music playlist queue loop

        if event.type == p.QUIT:  ###### Event ######
            running = False
        elif event.type == p.MOUSEBUTTONDOWN:
            location = p.mouse.get_pos()         # (x, y) location of mouse
            col = location[0] // SQ_SIZE
            row = location[1] // SQ_SIZE
            if sqSelected == (row, col):         # the user clicked the same square twice
                sqSelected = ()                  # deselect
                playerClicks = []                # clear player clicks
            else:
                sqSelected = (row, col)
                playerClicks.append(sqSelected)  # append for both 1st and 2nd clicks

            if len(playerClicks) == 2:  # after 2nd click
                move = ChessEngine.Move(playerClicks[0], playerClicks[1], game_state.board)
                print(move.GetChessNotation())
                game_state.makeMove(move)
                sqSelected = ()         # reset user clicks
                playerClicks = []
```

- **Adding Mouse-Click functionality in the Main driver, making sure that the mouse click location is relative, in case we add borders or side panels later.**

```
class Move():
    """Chess Moves"""  # keep track of each information, grab data from board
    # dictionary to map keys to values
    # key : value
    ranksToRows = {"1": 7, "2": 6, "3": 5, "4": 4,
                   "5": 3, "6": 2, "7": 1, "8": 0}
    rowsToRanks = {v: k for k, v in ranksToRows.items()}
    filesToCols = {"a": 0, "b": 1, "c": 2, "d": 3,
                   "e": 4, "f": 5, "g": 6, "h": 7}
    colsToFiles = {v: k for k, v in filesToCols.items()}

    def __init__(self, startSq, endSq, board):
        self.startRow = startSq[0]
        self.startCol = startSq[1]
        self.endRow = endSq[0]
        self.endCol = endSq[1]
        self.pieceMoved = board[self.startRow][self.startCol]  # keep track of what pieces are moved
        self.pieceCaptured = board[self.endRow][self.endCol]  # keep track of what pieces are being captured

    def GetChessNotation(self):
        # you can add to make this like real chess notation
        return self.GetRankFile(self.startRow, self.startCol) + self.GetRankFile(self.endRow, self.endCol)

    def GetRankFile(self, r, c):
        return self.colsToFiles[c] + self.rowsToRanks[r]
```

-

- **Keeping track of the chess moves information in ChessEngine**
- **Add columns & rows notations, in terms of chess.**

- Understanding chess notation

8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1
	a	b	c	d	e	f	g	h

- **Python 8x8 Matrix state: Row (0-7) top to bottom, Column (0-7) Left to Right, start at (0, 0) Top Left corner**
- **In Chess, Rows are called Ranks (1-8), while Columns are called Files (a-h).**
- **For instance, at (0,0) the computer notation will stored as "bR"(Black Rook), but in chess, we would say that the location is at "a8"**
- **So we store each key as value in the dictionary, to keep track of each chess move log. (line 40-45)**

Chess Notation move Log: Pawn from a2 > a3, Knight from g1 > f3, Pawn from h2 > h3

Rows notation from bottom to top: 1 > 8

Columns notation from LHS to RHS: A > H

- **Chess move test**

In this video, we can see that the chess piece was able to move by clicking on the piece and on the designated sq empty space. However, it was still an invalid move and a minor bug.

Invalid move & piece bug.mp4

Bugs:
If you click on an empty square then on the piece, the piece disappears. So I make an adjustment in makeMove():

Bug detected: Before

```
def makeMove(self, move):
    self.board[move.startRow][move.startCol] = "--"
    self.board[move.endRow][move.endCol] = move.pieceMoved
    self.moveLog.append(move)       # Log the move so we can undo it later,
    self.whiteToMove = not self.whiteToMove     # swap players
```

Bug-fixes: After

```
29      def makeMove(self, move):
30          if self.board[move.startRow][move.startCol] != "--":
31              self.board[move.startRow][move.startCol] = "--"
32              self.board[move.endRow][move.endCol] = move.pieceMoved
33              self.moveLog.append(move)  # Log the move so we can undo
```

No more disappearing chess pieces when missed clicks.

Disappearing piece bug fixes.mp4

24/10/21

Adding additional Ui features

```
def main():
    p.mixer.pre_init(44100, -16, 2, 2048)  # setup mixer to avoid sound lag
    p.init()
    p.mixer.init()  # Music mixer
    playlist = list()  # Music playlist
    playlist.append("music/Giorno's Theme in the style of JAZZ.mp3")
    playlist.append("music/FKJ - Ylang Ylang.mp3")  # 3
    playlist.append("music/FKJ - Die With A Smile.mp3")  # 2
    playlist.append("music/Wii.mp3")  # 1
    p.mixer.music.load(playlist.pop())  # Get the first track from the playlist
    p.mixer.music.queue(playlist.pop())  # Queue the 2nd song
    p.mixer.music.set_endevent(p.USEREVENT)  # Setup the end track event
    p.mixer.music.play()
    p.event.wait()
```

- **Adds multiple background music in playlist form.**

```
############## Main Chess ui : Don't touch ##################
screen = p.display.set_mode((1000, HEIGHT))  # (0,0) top left >>> (120
p.display.set_caption("|ChessM8| Checkmate Chess beta 1.0_by: Jirapat_
clock = p.time.Clock()
screen.fill(p.Color("white"))
game_state = ChessEngine.GameState()  # Calls _init_ from ChessEngine
loadImages()  # do this once, before while loop

############ Load on-screen images ######################
Kmitl = (p.image.load("Logo/KMITL-GO 200px.png"))
screen.blit(Kmitl, (800, 0))
FE = (p.image.load("Logo/FE Logo fix.png"))
screen.blit(FE, (800, 30))
```

- **Customize side panels**

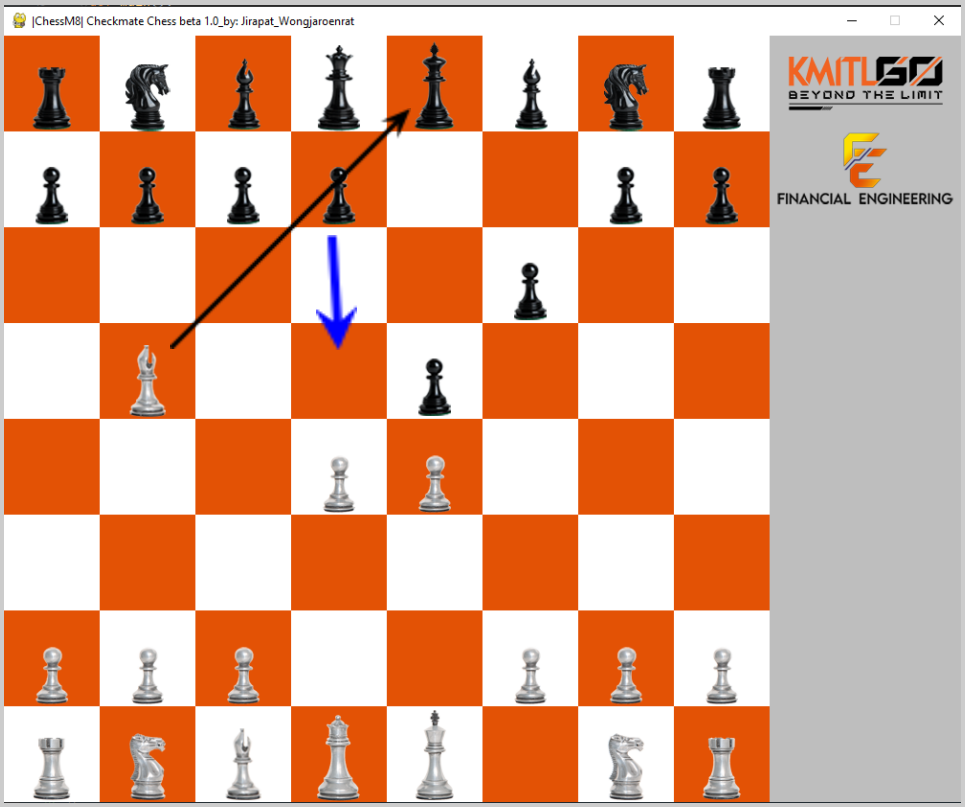
Output Ui

https://drive.google.com/file/d/1wEbgefSlJXZokai2vPnpkyrfSb6m1wPd/view?usp=sharing

26/10/21

Making legal chess move algorithm

- **Anti-cheat, generating all possible valid moves with/without considering checks.**



Consider **black** moving their **pawn** from **d7 to d6** or **d5.** Both are legal **"pawn moves"** but they are both invalid because they would leave the **black king** in check from the **white bishop** on **b5.**

For this reason, we will make the distinction between all **possible moves** and all **valid moves.**

So the basic algorithm for our getValidMoves() method will be like this:

➔ Get all possible moves

➔ For each possible move, check to see if it is a valid move by doing the following**:**

- Make the move
- Generate all possible moves for the opposite player
- See if any of the moves attacks your **king**.
- If your **king is safe**, it is a **valid move** and add it to a lis

➔ Return the list of valid moves only

Chess notation reference

8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1
	a	b	c	d	e	f	g	h

- **Preparing chess piece functions**

```
"""
  Get all the Knight moves for the pawn located at row, column and add these moves to the list
"""

def getKnightMoves(self, r, c, moves):
    pass

"""
  Get all the Bishop moves for the pawn located at row, column and add these moves to the list
"""

def getBishopMoves(self, r, c, moves):
    pass

"""
  Get all the Queen moves for the pawn located at row, column and add these moves to the list
"""

def getQueenMoves(self, r, c, moves):
    pass

"""
  Get all the King moves for the pawn located at row, column and add these moves to the list
"""

def getKingMoves(self, r, c, moves):
    pass
```

- **Undo Chess Move**

In ChessEngine

```
    """
    Undo the last move made
    """
    def undoMove(self):
        if len(self.moveLog) != 0:   # Make sure that there is a move to undo
            move = self.moveLog.pop()
            self.board[move.startRow][move.startCol] = move.pieceMoved
            self.board[move.endRow][move.endCol] = move.pieceCaptured
            self.whiteToMove = not self.whiteToMove # switch turns back
```

In the Main driver while loop, under User input.
When pressed "Z" key, undo chess move

```
            # key handlers
            elif event.type == p.KEYDOWN:
                if event.key == p.K_z:  # undo when "z" is being pressed
                    game_state.undoMove()
```

Pawn algorithm test + "z" key to undo moves
https://drive.google.com/file/d/1ejUk9iY_vGO8q1lT0YBkgyMObzdpBYVE/view?usp=sharing

***Note: Undo Move happens when valid moves** `def getValidMoves(self):`

31/10/2021

Chess Move Algorithm (continued):

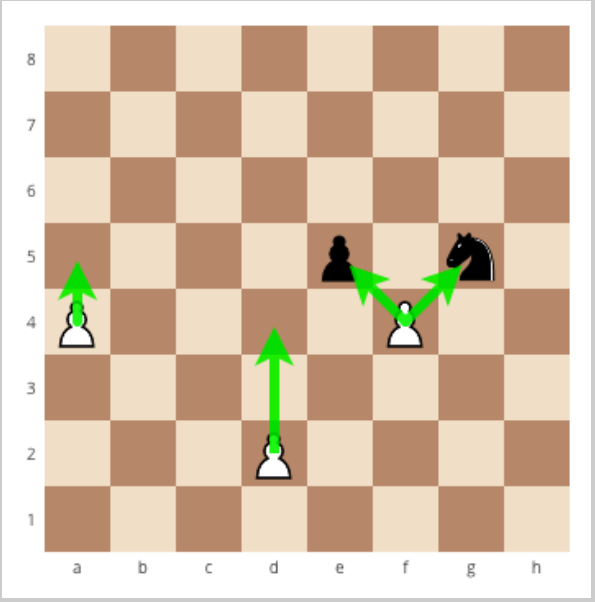

- Pawn Moves Logic
  - **Move exactly one square forward (can't move backwards)**
  - **Pawn may advance 2 squares forward on the first time it is moved.**
  - **Pawn can only capture (eats) 2 sq diagonally forward (Left or Right).**

```
"""
Get all the Pawn moves for the pawn located at row, column and add these moves to the list
"""

def getPawnMoves(self, r, c, moves):
    if self.whiteToMove:  ######## White pawn moves ############
        if self.board[r - 1][c] == "--":  # 1 square pawn advance
            moves.append(Move((r, c), (r - 1, c), self.board))
            if r == 6 and self.board[r - 2][c] == "--":  # 2 square pawn advance
                moves.append(Move((r, c), (r - 2, c), self.board))
        if c - 1 >= 0:  # Capture to the left (eats diagonally)
            if self.board[r - 1][c - 1][0] == "b":  # enemy piece to capture
                moves.append(Move((r, c), (r - 1, c - 1), self.board))
        if c + 1 <= 7:  # Captures to the right (eats diagonally)
            if self.board[r - 1][c + 1][0] == "b":  # enemy piece to capture
                moves.append(Move((r, c), (r - 1, c + 1), self.board))
```

```
95          else: # black pawn moves
96              if self.board[r + 1][c] == "--":  # 1 square move
97                  moves.append(Move((r, c), (r + 1, c), self.board))
98                  if r == 1 and self.board[r + 2][c] == "--":  # 2 square pawn move
99                      moves.append(Move((r, c), (r + 2, c), self.board))
100
101             if c - 1 >= 0:  # Capture to the left (eats diagonally)
102                 if self.board[r + 1][c - 1][0] == "w":
103                     moves.append(Move((r, c), (r + 1, c - 1), self.board))
104
105             if c + 1 <= 7:  # Captures to the right (eats diagonally)
106                 if self.board[r + 1][c + 1][0] == "w":
107                     moves.append(Move((r, c), (r + 1, c + 1), self.board))
```

```
self.board = [
    ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],
    ["bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"],
    ["--", "--", "--", "--", "--", "--", "--", "--"],
    ["--", "--", "--", "--", "--", "--", "--", "--"],
    ["--", "--", "--", "--", "--", "--", "--", "--"],
    ["--", "--", "--", "bp", "--", "--", "--", "--"],
    ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],
    ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"],
]
```

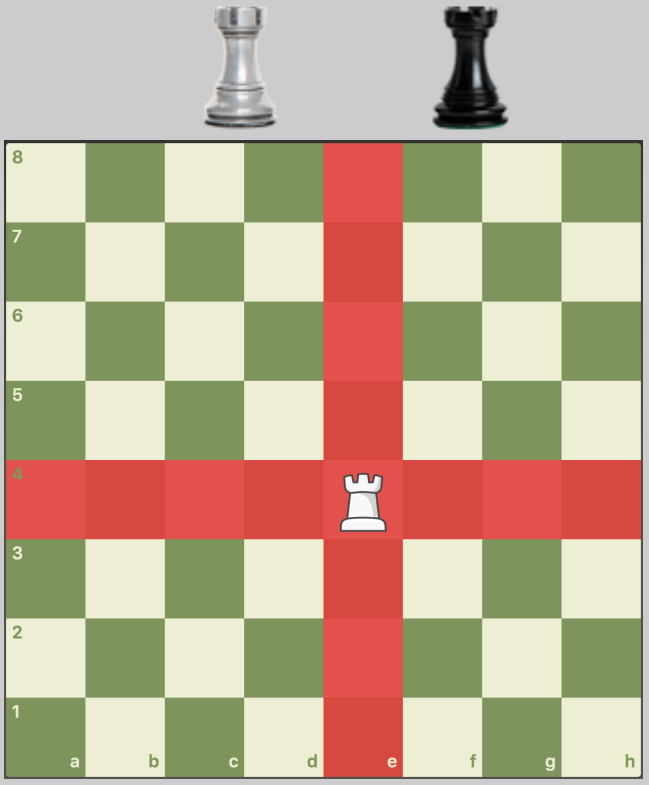
Replace black pawn 'bp' on blank square on chess gamestate for dummy white pawn capture



https://drive.google.com/file/d/1QEeJpcop1mIhCDp9CNzTWY_60dxBpKc9/view?usp=sharing

2/11/2021

Chess Move Algorithm (continued):



- Rook Moves Logic
  - **Rook moves as many squares as it likes left or right horizontally, or as many squares as it likes up or down vertically, + plus-sign pattern (up-down-left-right).**
  - **It cannot move across a piece.**
  - **Rook can move until it ran into a piece**

In python, we can define directions patterns that rook can move up to 7 empty squares. So we only check for 7 squares in 4 directions.

Define directions patterns example:

(-1, 0) = -1 row & 0 column is moving **upwards** by 1 square

```
114     def getRookMoves(self, r, c, moves):
115         directions = ((-1, 0), (0, -1), (1, 0), (0, 1))  #up, left, down, right | defir
116         enemyColor = "b" if self.whiteToMove else "w"
117         if self.whiteToMove:
118             enemyColor = "b"
119         else:
120             enemyColor = "w"
121         for d in directions:
122             for i in range(1, 8):
123                 endRow = r + d[0] * i
124                 endCol = c + d[1] * i
125                 if 0 <= endRow < 8 and 0 <= endCol < 8:     # make sure if on board
126                     endPiece = self.board[endRow][endCol]
127                     if endPiece == "--":            # check when rook runs into a piece
128                         moves.append(Move((r, c), (endRow, endCol), self.board))
129                     elif endPiece[0] == enemyColor:  # enemy piece valid move
130                         moves.append(Move((r, c), (endRow, endCol), self.board))
131                         break
132                     else:  # friendly piece invalid
133                         break
134                 else:  # off board
135                     break
```

Rook move test

https://drive.google.com/file/d/1gBk8dHNpaFGRSFc5QxyGhjv2s72s0TVb/view?usp=sharing

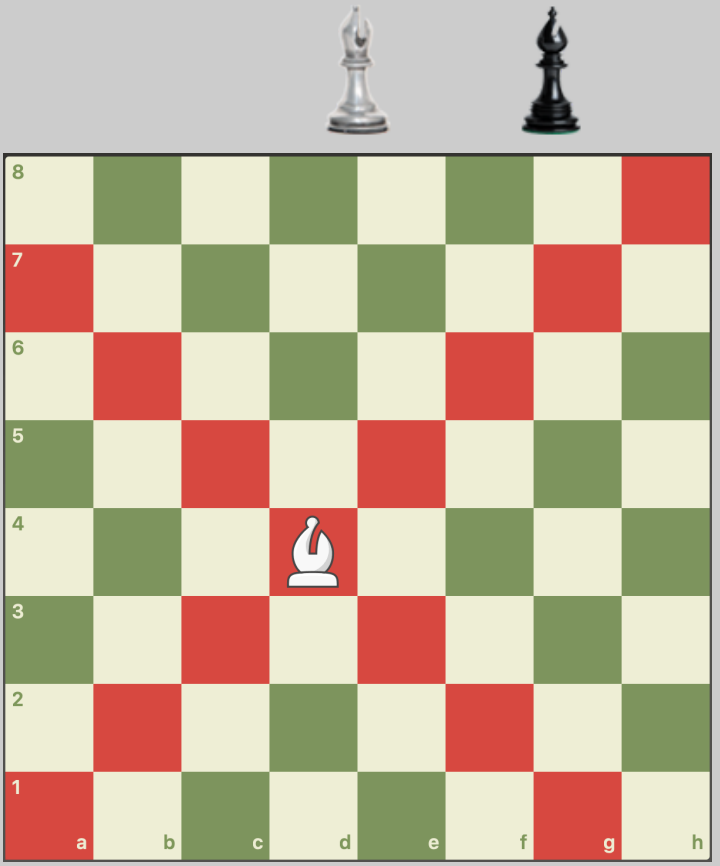
```
enemyColor = "b" if self.whiteToMove else "w"
```

Is the same as

```
if self.whiteToMove:
    enemyColor = "b"
else:
    enemyColor = "w"
```

5/11/21

- Bishop Moves Logic
  - **Similar to Rook, Bishop moves as many square as it likes in a given direction, in an "X" pattern diagonally (upwards, downwards), left and right.**
  - **It cannot move across a piece.**
  - **Bishop can move until it ran into a piece**



In python, we can define directions patterns that bishop can move up to 7 empty squares diagonally.

So we list out the coordinates of where they can move in 4 potential directions.

Define directions patterns example:

- **(-1, -1) = -1 row & -1 column** is moving **Upwards diagonally to the Left**
- **(-1, 1) = -1 row & 1 column** is moving **Upwards diagonally to the Right**

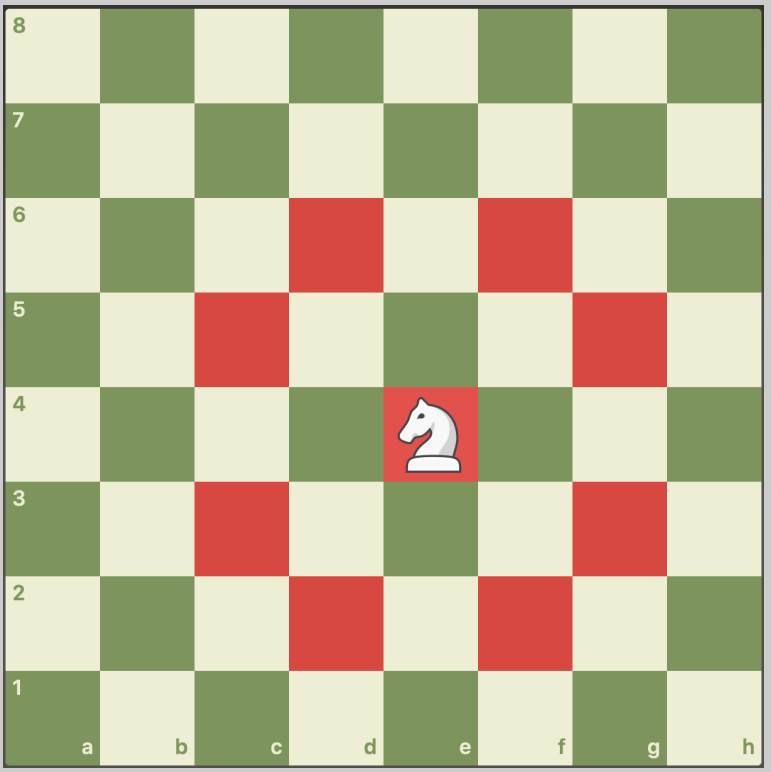
Bishop & Rook are similar, changing only directions...

```
144     def getBishopMoves(self, r, c, moves):
145         directions = ((-1, -1), (-1, 1), (1, -1), (1, 1))  # 4 diagonals
146         enemyColor = "b" if self.whiteToMove else "w"
147         for d in directions:
148             for i in range(1, 8): # bishop can move 7 squares maximum
149                 endRow = r + d[0] * i
150                 endCol = c + d[1] * i
151                 if 0 <= endRow < 8 and 0 <= endCol < 8:  # make sure if on board
152                     endPiece = self.board[endRow][endCol]
153                     if endPiece == "--":  # check when rook runs into a piece
154                         moves.append(Move((r, c), (endRow, endCol), self.board))
155                     elif endPiece[0] == enemyColor:  # enemy piece valid move
156                         moves.append(Move((r, c), (endRow, endCol), self.board))
157                         break
158                     else:  # friendly piece invalid
159                         break
160                 else:  # off board
161                     break
162
```

Bishop Move Test
https://drive.google.com/file/d/1Hhi_9MYo3EKrhQYv-sM8rS-euZIgSTR-/view?usp=sharing

- Knight Moves Logic
  - **The knight moves in an "L-shape." (move 8 sq directions around itself)**
  - **It moves one square left or right horizontally and then two squares up or down vertically, or it moves two squares left or right horizontally and then one square up or down vertically.**

  - **The knight is the only piece in chess that can jump over another piece**
  - **The knight can capture only what it lands on, not what it jumps over**



In python, we can define direction patterns of the Knight moves by listing out the coordinates of where it can move.

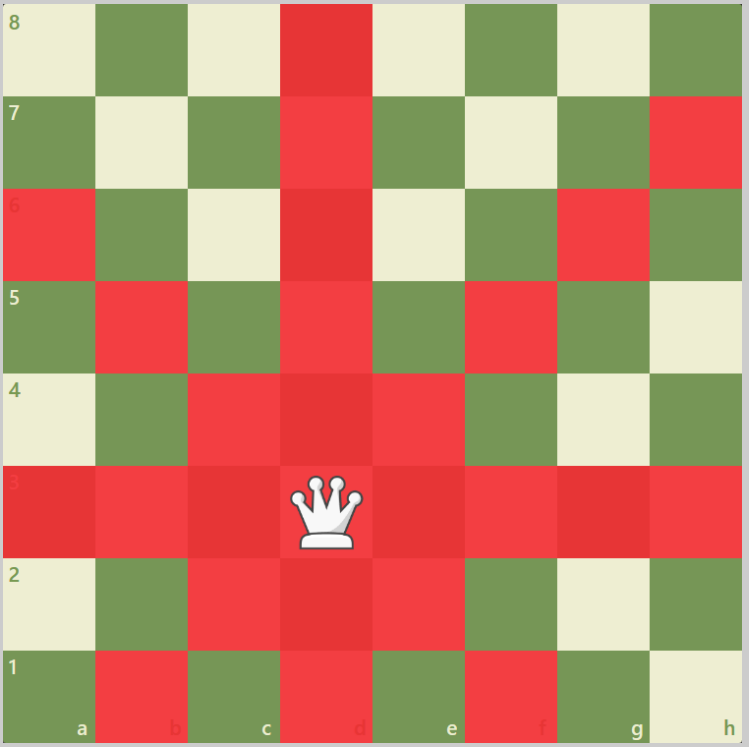
Define knightMoves example:

- **(-2, -1) = -2 row & -1 column** is moving **2 square Upwards & 1 square Left**
- **(-2, 1) = -2 row & 1 column** is moving **2 square Upwards & 1 square Right**
- **(-1, -2) = -1 row & -2 column** is moving **1 square Upwards & 2 square Left**
- **(-1, 2) = -1 row & 2 column** is moving **1 square Upwards & 2 square Right**

```
133     """
134         Get all the Knight moves for the pawn located at row, column and add these moves to the list
135     """
136
137     def getKnightMoves(self, r, c, moves):
138         knightMoves = ((-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1))
139         allyColor = "w" if self.whiteToMove else "b"
140         for d in knightMoves:
141             endRow = r + d[0]
142             endCol = c + d[1]
143             if 0 <= endRow < 8 and 0 <= endCol < 8:
144                 endPiece = self.board[endRow][endCol]
145                 if endPiece[0] != allyColor:  # not an ally piece (empty or enemy piece)
146                     moves.append(Move((r, c), (endRow, endCol), self.board))
147
```

- Queen Moves Logic
  - **The Queen is considered a major piece, it moves like a rook and a bishop combined**

  - **It can move as many squares as it likes left or right horizontally, or as many squares as it likes up or down vertically (like a rook).**
  - **The queen can also move as many squares as it likes diagonally (like a bishop)**



In Python, since we've already defined both Rook & Bishop moves. Where it's just a rook & bishop combined.

We can come up with a clever solution without writing any additional codes. But only by calling in those Rook & Bishop functions.

```
176
177         # Bishop x Rook combined, smart method
178         def getQueenMoves(self, r, c, moves):
179             self.getRookMoves(r, c, moves)
180             self.getBishopMoves(r, c, moves)
181
```

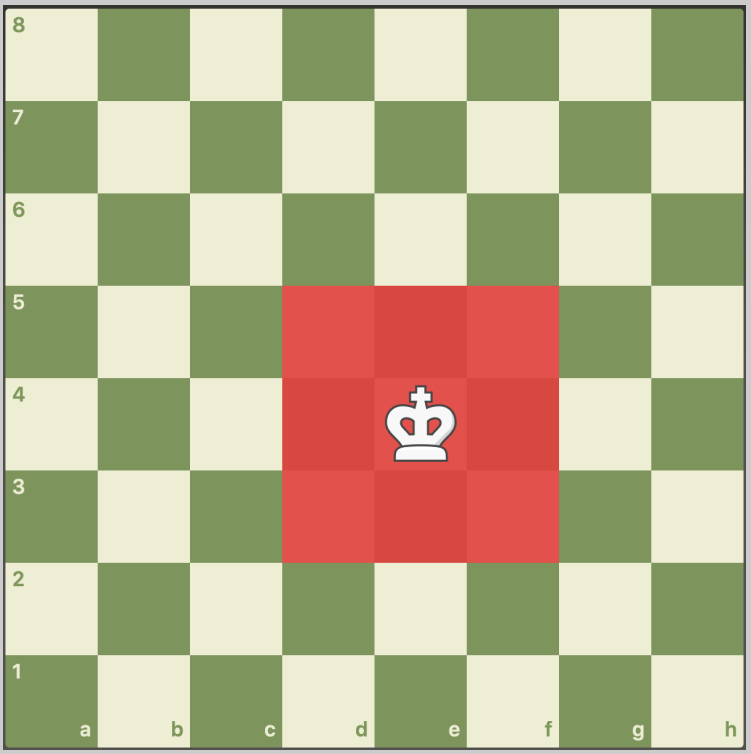
Queen move test
https://drive.google.com/file/d/1Ma1LlL3lbxk6lkZuIpgBtLgnUKXNOFMu/view?usp=sharing

- King Moves Logic

- **The King is the most important chess piece, it moves 1 square in any direction.**

- **The king is not a very powerful piece, as it can only move (or capture) one square in any direction. Note that the king cannot be captured.**

- **The goal of a game of chess is to checkmate the king. When a king is attacked, it is called "check."**



In python, we can define direction patterns of the Knight moves by listing out the coordinates of where it can move by 8 squares around itself.

Define knightMoves example:

- **(-1, -1) = -1 row & -1 column** is moving **1 sq. Upwards & 1 sq. to the Left**
- **(-1, 0) = -1 row & 0 column** is moving **1 sq. Upwards**
- **(-1, 1) = -1 row & 1 column** is moving **1 sq. Upwards & 1 sq. to the Left**
- **(0, -1) = 0 row & -1 column** is moving **1 sq. Downwards**

```
182     """
183         Get all the King moves for the pawn located at row, column and add these moves to the list
184     """
185
186     def getKingMoves(self, r, c, moves):
187         kingMoves = ((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1))
188         allyColor = "w" if self.whiteToMove else "b"
189         for i in range(8):
190             endRow = r + kingMoves[i][0]
191             endCol = c + kingMoves[i][1]
192             if 0 <= endRow < 8 and 0 <= endCol < 8:
193                 endPiece = self.board[endRow][endCol]
194                 if endPiece[0] != allyColor:  # not an ally piece (empty or enemy piece)
195                     moves.append(Move((r, c), (endRow, endCol), self.board))
```

King move test

https://drive.google.com/file/d/1kwUQAPrigvyP9974GtX92vzirU430eMQ/view?usp=sharing

7/11/21

Chess User Input: Minor bug fix

Before: When the player clicks the first piece, but then decides to move the other piece instead. The game then registered it as the second click which goes off before making the move. Forcing the player to click on the new piece again.

```
if len(playerClicks) == 2:  # after 2nd click
    move = ChessEngine.Move(playerClicks[0], playerClicks[1], game_state.board)
    print(move.GetChessNotation())
    game_state.makeMove(move)
    sqSelected = ()         # reset user clicks
    playerClicks = []
```

https://drive.google.com/file/d/10OCTpl7TXVVVfZ3wmev4YyHP45tv23Go/view?usp=sharing

```
103
104                 if len(playerClicks) == 2:  # after 2nd click
105                     move = ChessEngine.Move(playerClicks[0], playerClicks[1], game_state.board)
106                     print(move.GetChessNotation())
107                     if move in validMoves:
108                         game_state.makeMove(move)
109                         moveMade = True
110                         sqSelected = ()  # reset user clicks
111                         playerClicks = []
112                     else:
113                         playerClicks = [sqSelected]
```

After: By indenting **sqSelected** and **playerClicks** and adding else conditional Allow us to fix the error, players can now do only 3 clicks to change and register a new piece.

(1st piece click - 2nd piece click when change their mind - 3rd click to user move input)

https://drive.google.com/file/d/1SKFbdLSblwoCmTa7GVUhWkzwYmRDqzns/view?usp=sharing

11/11/21

Chess Moves Algorithm: Checks, Checkmate and Stalemate

- **What is Check?**
  When a **king** is attacked, it is called check (written as "+" in chess notation). Check can be viewed as saying "The king is being attacked!" Since a king can never be captured, the term **"check" is used when a king is threatened.**

- **What is Checkmate?**
  When a **king** is attacked, it is called a check. A checkmate occurs when a king is placed in check and has no legal moves to escape. When a checkmate happens, the game ends immediately, and the player who delivered the checkmate wins.

- **Stalemate is a kind of draw** that happens when one side has NO legal moves to make. If the king is NOT in check, but no piece can be moved without putting the king in check, then the game will end with a stalemate draw.

```
    def getValidMoves(self):
        #1) Get a List of all possible Moves
        moves = self.getAllPossibleMoves()
        #2) Make a move from the list of possible moves
        for i in range(len(moves)-1, -1, -1): # when removing elements from a list go backwards through that list
            self.makeMove(moves[i])
            self.whiteToMove = not self.whiteToMove
        #3) Generate all of the opponents move after making the move in the previous
        #4) Check if any of the opponents move attacks your king -> if so remove the moves from our list
            if self.inCheck():
                moves.remove(moves[i])
                print("Check White" if self.whiteToMove else "Check Black")
            self.whiteToMove = not self.whiteToMove
            self.undoMove()
        #5) Return the final list of moves
        if len(moves) == 0:     # either checkmate or stalemate
            if self.inCheck():
                print("CHECK MATE! " + ("White" if not self.whiteToMove else "Black") + " wins.")
                self.checkMate = True
            else:
                print("Draw, due to STALEMATE")
                self.staleMate = True

        else:
            self.checkMate = False
            self.staleMate = False

        return moves
```

```
#Update the KING's Position
if move.pieceMoved == "wK":
    self.whiteKingLocation = (move.startRow, move.startCol)
elif move.pieceMoved == "bK":
    self.blackKingLocation = (move.startRow, move.startCol)
```

```
#Keeping track of the Kings to make valid move calculation and castling easier.
self.whiteKingLocation = (7, 4)
self.blackKingLocation = (0, 4)

#Keep track of checkmate and stalemate
self.checkMate = False
self.staleMate = False
```

After generating the checks algorithm, when the king has been checked.
The program will only allow valid moves that will block the king from being attacked. When there are no valid moves available, the game will end with checkmate!

For more understanding, please watch this video demonstration:

Check & Checkmate test
https://drive.google.com/file/d/17qFnu_dGUZy1_vn0MO5irAykTGkb29DF/view?usp=sharing

14/11/21

Chess Moves Algorithm: Checks, Checkmate and Stalemate (Continued)

White-turn to move

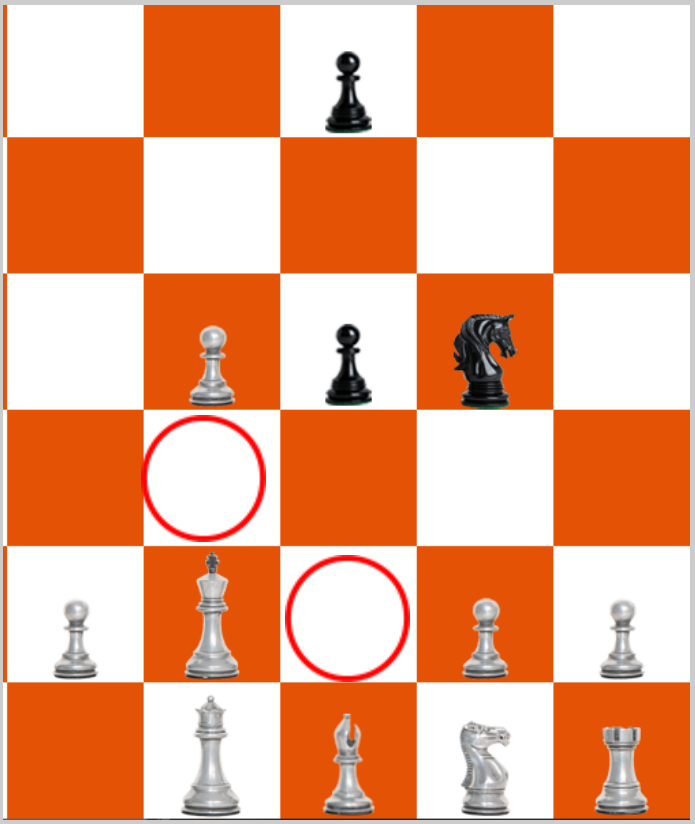
In every chess move, the algorithm will check for a valid King move (is the King being attacked)?

In this image it is **White-turn to move**, the player won't be able to move the Pawn that is in a red circle.

This is because the chess algorithm has already run checks of all of the black pieces possible moves that will make the **White King** to be on Check or under attack.

In this scenario, the **black Queen** will check the king if that **White Pawn** moves from that square. So the program didn't allow the player to move it. This is called a
"PING", an any piece move to prevent/block its King from getting captured (Check)

White-turn to move



Another example, it's **the White-turn** to move again. In this image, the **White King** cannot move to the squares that are circled in red.

The chess algorithm will check all the player's white pieces valid moves that won't make its **White King** to be Check.

Therefore, players will be allowed to do any valid chess moves aside from moving the **White King** to those two squares that are circled in red.

16/11/21

Chess Move Advance-Algorithm: Pawn promotion & special pawn capture (En-passant)

To check whether this move is a pawn promotion, true or false?

ChessEngine: Edit the init function in `class Move` and add the Pawn Promotion logic (line 291-294). When White Pawn reach row 0 on the opponent's back side `self.pieceMoved == "wp" and self.endRow == 0`, as well as row 7 `self.pieceMoved == "bP" and self.endRow == 7` on our side.

```
284     def __init__(self, startSq, endSq, board):
285         self.startRow = startSq[0]
286         self.startCol = startSq[1]
287         self.endRow = endSq[0]
288         self.endCol = endSq[1]
289         self.pieceMoved = board[self.startRow][self.startCol]  # keep track of what pieces are moved, can't b
290         self.pieceCaptured = board[self.endRow][self.endCol]  # keep track of what pieces was captured, can b
291         # ----Pawn Promotion----#
292         self.isPawnPromoted = False
293         if (self.pieceMoved == "wp" and self.endRow == 0) or (self.pieceMoved == "bP" and self.endRow == 7):
294             self.isPawnPromoted = True
295 
296         self.moveID = self.startRow * 1000 + self.startCol * 100 + self.endRow * 10 + self.endCol
```

In Line (57-78), `move.isPawnPromoted` to check whether the pawn has reached the back rank. `self.board[move.endRow][move.endCol] = move.pieceMoved[0]`, to grab the color of the piece moved. The player will have a choice to choose which chess piece they wanted to be promoted to. At this point, I'll append "Q" as a hardcoding promotion to Queen. Promotion choice parameters will be added later.

```
45     def makeMove(self, move):
46         self.board[move.startRow][move.startCol] = "--"  # --Empty the start
47         self.board[move.endRow][move.endCol] = move.pieceMoved  # Keep the p
48         self.moveLog.append(move)  # Log/record the move, "Or replay history
49         self.whiteToMove = not self.whiteToMove  # swap players turn
50         # Update the King's Position if moved
51         if move.pieceMoved == "wK":
52             self.whiteKingLocation = (move.endRow, move.endCol)
53         elif move.pieceMoved == "bK":
54             self.blackKingLocation = (move.endRow, move.endCol)
55 
56         #----Pawn Promotion----#
57         if move.isPawnPromoted:
58             self.board[move.endRow][move.endCol] = move.pieceMoved[0] + "Q"
```

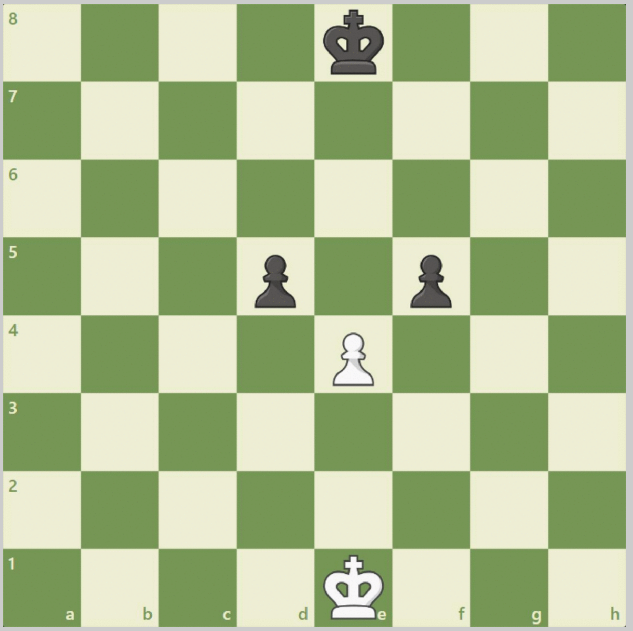
Pawn Promotion Test
https://drive.google.com/file/d/1VYwGzJvXms0eP-dxH7oP7fRTWoZ6Bc33/view?usp=sharing

- ## What Is The En-Passant Rule?

The en passant rule is a special pawn capturing move in chess. "En passant" is a French expression that translates to "in passing," which is precisely how this capture works.

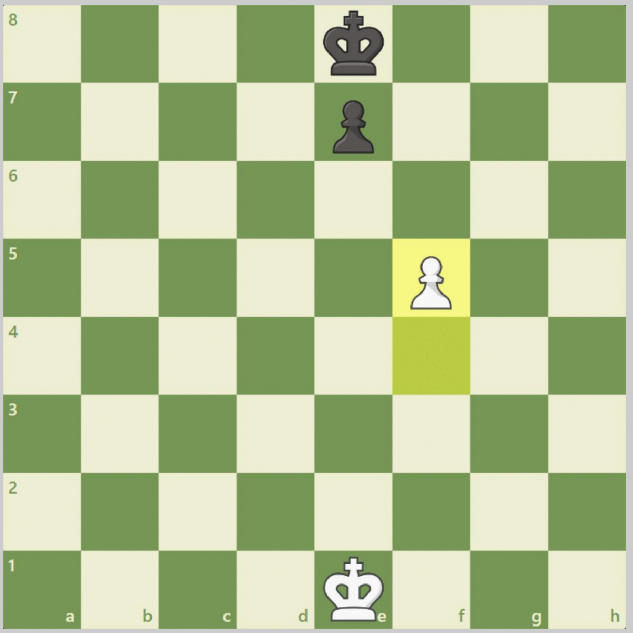
Pawns can usually capture only pieces that are directly and diagonally in front of it on an adjacent file. It moves to the captured piece's square and replaces it.

This is the regular way a pawn can capture another piece



There are a few requirements for the move to be legal:

1. The capturing pawn (white) must have advanced exactly three ranks to perform this move.
2. The **captured pawn (black)** must have moved two squares in one move, landing right next to the capturing pawn.
3. The en-passant capture must be performed on the turn immediately after the pawn being captured moves. If the player does not capture en-passant on that turn, they no longer can do it later.



This is how Pawn captures en-passant

Reference:

https://www.chess.com/terms/en-passant

En-Passant logic, line (297-299)

```
286     def __init__(self, startSq, endSq, board, isEnPassantMove = False):
287         self.startRow = startSq[0]
288         self.startCol = startSq[1]
289         self.endRow = endSq[0]
290         self.endCol = endSq[1]
291         self.pieceMoved = board[self.startRow][self.startCol]  # keep track of what pieces are mov
292         self.pieceCaptured = board[self.endRow][self.endCol]  # keep track of what pieces was capt
293         # ----Pawn Promotion----#
294         self.isPawnPromotion = (self.pieceMoved == "wp" and self.endRow == 0) or (self.pieceMoved
295 
296         #---En-Passant--- #
297         self.isEnPassantMove = isEnPassantMove
298         if self.isEnPassantMove:
299             self.pieceCaptured = 'wp' if self.pieceMoved == 'bP' else 'bp'
300 
```

Create en-passant capture move in `def getPawnMoves`

`(166-167),(172-173),(183-184), (190-191)` Append the exact same pawn move that we had before, the only difference is it's not a normal move, add an additional parameter from the init function.

```
155     def getPawnMoves(self, r, c, moves):
156         # ------ White pawn moves ------#
157         if self.whiteToMove and self.board[r][c][0] == "w":
158             if self.board[r - 1][c] == "--":  # 1 square pawn advance
159                 moves.append(Move((r, c), (r-1, c), self.board))
160                 if r == 6 and self.board[r - 2][c] == "--":  # 2 square pawn advance
161                     moves.append(Move((r, c), (r-2, c), self.board))
162             # captures
163             if c - 1 >= 0:  # Capture to the left (diagonally)
164                 if self.board[r - 1][c - 1][0] == "b":  # enemy piece to capture
165                     moves.append(Move((r, c), (r-1, c-1), self.board))
166                 elif (r - 1, c - 1) == self.enPassantPossible:
167                     moves.append(Move((r, c), (r-1, c-1), self.board, isEnPassantMove=True))
168 
169             if c + 1 <= 7:  # Captures to the right (diagonally)
170                 if self.board[r - 1][c + 1][0] == "b":  # enemy piece to capture
171                     moves.append(Move((r, c), (r-1, c+1), self.board))
172                 elif (r - 1, c + 1) == self.enPassantPossible:
173                     moves.append(Move((r, c), (r-1, c+1), self.board, isEnPassantMove=True))
174         else:  # -----black pawn moves-----#
175             if self.board[r + 1][c] == "--":  # 1 square move
176                 moves.append(Move((r, c), (r + 1, c), self.board))
177                 if r == 1 and self.board[r + 2][c] == "--":  # 2 square pawn move
178                     moves.append(Move((r, c), (r+2, c), self.board))
179             # captures
180             if c - 1 >= 0:  # Capture to the left (diagonally)
181                 if self.board[r + 1][c - 1][0] == "w":
182                     moves.append(Move((r, c), (r+1, c-1), self.board))
183                 elif (r + 1, c - 1) == self.enPassantPossible:
184                     moves.append(Move((r, c), (r+1, c-1), self.board, isEnPassantMove=True))
185 
186             if c + 1 <= 7:  # Captures to the right (diagonally)
187                 if self.board[r + 1][c + 1][0] == "w":
188                     moves.append(Move((r, c), (r+1, c+1), self.board))
189                 elif (r + 1, c + 1) == self.enPassantPossible:
190                     moves.append(Move((r, c), (r + 1, c + 1), self.board, isEnPassantMove=True))
```

En-Passant test
https://drive.google.com/file/d/1sfdfrqXGM1vyr-wzzPGepJ4qLPICUJcT/view?usp=sharing

20/11/21

Minor bug fix: Chess user clicks

Before: The bug I detect in the Chess gui is when the Player/Users clicks on the screen aside from the chess board location (i.e. the side panel on the left) then clicks on the chess piece to move after or vice versa. The game crashes.

Bug issue video
https://drive.google.com/file/d/1BawpJz7PJSpinslnPJfODuZmNBM9s5KT/view?usp=sharing

After: We add a condition to the Chess main driver/ Mouse handlers section than when the Users click out of board (on move log panel) to do nothing.

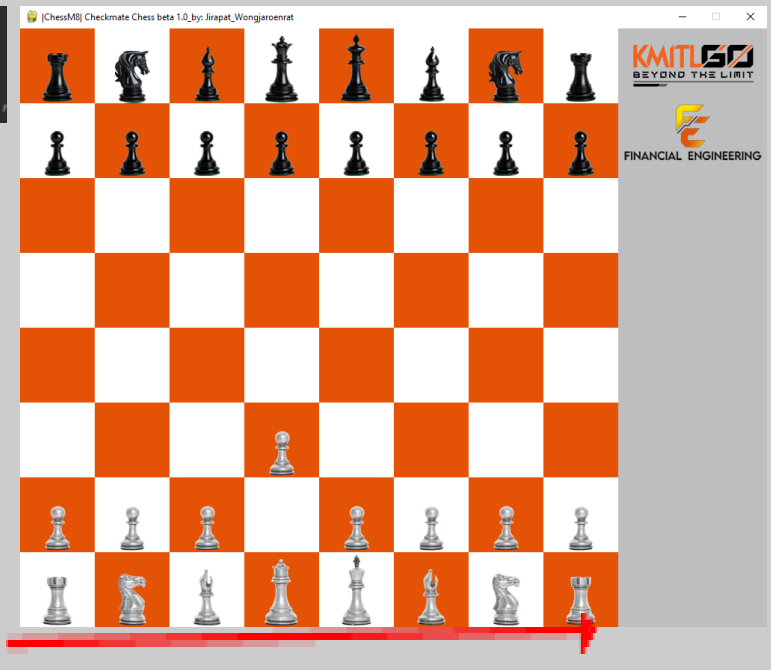
Since the **Height of the screen dimension matches the Row of the chess board,** we just only include the index range for the column of the chess board (0-8) to cover the user clicks to avoid the crash.

Do the Math!: The gui whole screen dimension (W=1000px x H=800px), chess board (8x8) x 100px = 800px user click range

```
94          ########## Mouse Handlers: User input ################
95          elif event.type == p.MOUSEBUTTONDOWN:
96              location = p.mouse.get_pos()  # (x, y) location of mouse
97              col = location[0] // SQ_SIZE
98              row = location[1] // SQ_SIZE
99              if (col >= 8) or col < 0:  # Click out of board (on move log panel) -> d
100                 continue
101             if sqSelected == (row, col):  # the user clicked the same square twice
102                 sqSelected = ()  # deselect
```

```
WIDTH = HEIGHT = 800  # 512, 400 another option
DIMENSION = 8  # dimensions of a chess board are 8x8
SQ_SIZE = HEIGHT // DIMENSION  # 100 x 100px per SQ
```

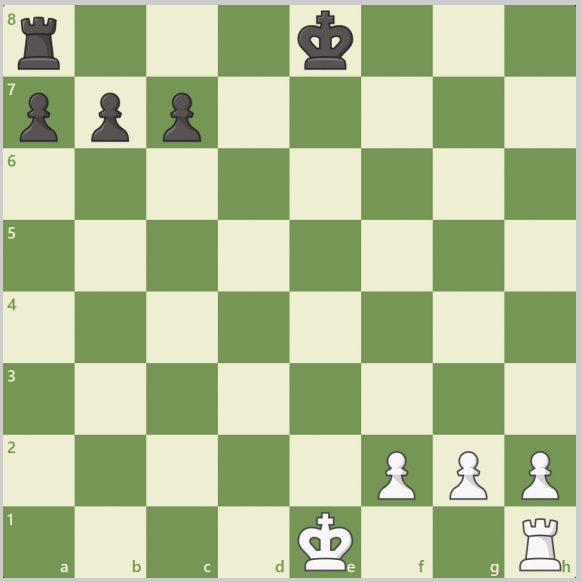
Outcome after bug fixes.
https://drive.google.com/file/d/18tan0Eg3b0y0wAqJP4kLTKfSqJks5g5N/view?usp=sharing



22/11/21

Chess Moves Advanced-Algorithm: Chess Castling

- ***What is Castling?***



Castling is a move in the game of chess involving a player's king and either of the player's original rooks. It is the only move aside from the knight's move where a piece can be said to "jump over" another.

This special move is the only time you can move two pieces in the same turn. Castling only involves the king and the rook (no other chess pieces)

- **Castling Requirements**

Rule 1. There are no pieces between the king and the chosen rook

Rule 2. You cannot castle if you have moved your king (or the rook)!

Rule 3. You are not allowed to castle through check!

Rule 4. You are not allowed to castle through check!

The black bishop on b4 is making a check. White cannot castle while in check.

White is not allowed to castle through the bishop's "check" on f1.

First, we set the current castling rights as variable and set its initial conditions to keep track.

```
# Castling
self.currentCastlingRights = CastleRights(True, True, True, True)
# self.castleRightsLog = [self.currentCastlingRights] # this will pose a problem as we are not c
# self.currentCastlingRights object we are just storing another reference to it.
self.castleRightsLog = [
    CastleRights(self.currentCastlingRights.wks, self.currentCastlingRights.wqs,  # correct way
                 self.currentCastlingRights.bks, self.currentCastlingRights.bqs)]
```

Add in **Castling class** for the True, False variables

```
class CastleRights():

    def __init__(self, wks, wqs, bks, bqs):
        self.wks = wks
        self.wqs = wqs
        self.bks = bks
        self.bqs = bqs

    def __str__(self):
        return ("7. Castling Rights(wk, wq, bk, bq): " + str(self.wks) + " " + str(self.wqs)
                + " " + str(self.bks) + " " + str(self.bqs))
```

wks/bks = **White/Black** king's side (right-castling)
wqs/bks = **White/Black** queen's side (left-castling)

```
def updateCastleRights(self, move):
    if move.pieceMoved == "wK":
        self.currentCastlingRights.wqs = False
        self.currentCastlingRights.wks = False

    elif move.pieceMoved == "bK":
        self.currentCastlingRights.bks = False
        self.currentCastlingRights.bqs = False

    elif move.pieceMoved == "wR":
        if move.startRow == 7 and move.startCol == 0:  # Left Rook
            self.currentCastlingRights.wqs = False
        if move.startRow == 7 and move.startCol == 7:  # Right Rook
            self.currentCastlingRights.wks = False

    elif move.pieceMoved == "bR":
        if move.startRow == 0 and move.startCol == 0: # Left Rook
            self.currentCastlingRights.bqs = False
        if move.startRow == 0 and move.startCol == 7: # Right Rook
            self.currentCastlingRights.bks = False
```

1. Setting Castling Rights, if you've moved your king or rook

2. Add in **Castle move algorithm** in `def makeMove` , consist of 2 parts. When the move is the Castling move, we have to change where the rook location was

3. After any moves we update the castle rights to check whether its still in requirements to do Castling move

```
# ---Castle Move--- #
if move.isCastleMove:
    if move.endCol < move.startCol:  # Queen side castle (left)
        self.board[move.endRow][0] = "--" # erase old rook
        self.board[move.endRow][move.endCol + 1] = move.pieceMoved[0] + "R" # move the rook
    else:  # King side castle (right)
        self.board[move.endRow][7] = "--" # erase old rook
        self.board[move.endRow][move.endCol - 1] = move.pieceMoved[0] + "R" # move the rook

# ---Update Castling Rights--- #
self.updateCastleRights(move)
newCastleRights = CastleRights(self.currentCastlingRights.wks, self.currentCastlingRights.wqs,
                               self.currentCastlingRights.bks, self.currentCastlingRights.bqs)
self.castleRightsLog.append(newCastleRights)
```

Add in 3 Castling functions. Generate all the Valid Castling moves for the King at row, column and add them to the list of moves. **Where there were 4 possible moves (wqs, wks, bqs, bks). Basically, this is where the Castling move algorithm is being added.**

```
391     """
392     Get all the King castling move for the king at row, column and add them to the list
393     """
394     def getCastlingMoves(self, r, c, moves):
395         if self.inCheck():
396             return  # can't castle when king is under attack
397 
398         if self.whiteToMove and self.currentCastlingRights.wks or \
399                 (not self.whiteToMove and self.currentCastlingRights.bks):
400             self.getKingSideCastleMoves(r, c, moves)
401 
402         if self.whiteToMove and self.currentCastlingRights.wqs or \
403                 (not self.whiteToMove and self.currentCastlingRights.bqs):
404             self.getQueenSideCastleMoves(r, c, moves)
405 
406     def getKingSideCastleMoves(self, r, c, moves):
407         if self.board[r][c + 1] == "--" and self.board[r][c + 2] == "--":
408             if not self.isUnderAttack(r, c + 1) and not self.isUnderAttack(r, c + 2):
409                 moves.append(Move((r, c), (r, c + 2), self.board, isCastleMove=True))
410 
411     def getQueenSideCastleMoves(self, r, c, moves):
412         if self.board[r][c - 1] == "--" and self.board[r][c - 2] == "--" and self.board[r][c - 3] == "--":
413             if not self.isUnderAttack(r, c - 1) and not self.isUnderAttack(r, c - 2):
414                 moves.append(Move((r, c), (r, c - 2), self.board, isCastleMove=True))
```

And finally, we then add Castling Rights & Castling Move in `def undoMove` function. To be able to undo the Castling moves.

```
125         # ---Undo Castling Rights--- #
126         self.castleRightsLog.pop()  # get rid of new Castling rights from the move we are undoing
127         self.currentCastlingRights.wks = self.castleRightsLog[-1].wks # set the current castle rights
128         self.currentCastlingRights.wqs = self.castleRightsLog[-1].wqs # update current castling right
129         self.currentCastlingRights.bks = self.castleRightsLog[-1].bks # update current castling right
130         self.currentCastlingRights.bqs = self.castleRightsLog[-1].bqs # update current castling right
131 
132         # ----Undo Castle Move---- #
133         if move.isCastleMove:
134             if move.endCol < move.startCol:  # Queen side castle (left)
135                 self.board[move.endRow][move.endCol + 1] = "--" # remove rook
136                 self.board[move.endRow][0] = move.pieceMoved[0] + "R" # replace rook
137             else:  # King side castle (right)
138                 self.board[move.endRow][move.endCol - 1] = "--" # remove rook
139                 self.board[move.endRow][7] = move.pieceMoved[0] + "R" # replace rook
```

Castling moves & undo moves Test
https://drive.google.com/file/d/1wa6q-3r_vgTOxHUodRiwsA_opb6yznCF/view?usp=sharing

9/12/21

Chess Main: Adding User Interface Features

- *Highlighting Chess piece moves*

_ When a player clicks on a piece it's going to highlight the piece that has been selected with one color and it'll also highlight all the squares of possible moves that it can make with another color. Then if you click on a different piece, it'll show the current ones instead. Can be used as a debugging tool because you can see if it highlights the square that you don't think that it should be able to move to.

```
def highlightSquares(screen, game_state, validMoves, sqSelected):
    if sqSelected != ():
        r, c = sqSelected
        # sqSelected is a piece that can be moved
        if game_state.board[r][c][0] == ("w" if game_state.whiteToMove else "b"):
            #highlight selected square
            s = p.Surface((SQ_SIZE, SQ_SIZE))
            s.set_alpha(230) # transparency value -> 0 transparent; 255 opaque
            s.fill(p.Color("blue"))
            screen.blit(s, (c*SQ_SIZE, r*SQ_SIZE))
            #highlight moves from that square
            s.fill(p.Color("yellow"))
            for move in validMoves:
                if move.startRow == r and move.startCol == c:
                    screen.blit(s, (move.endCol*SQ_SIZE, move.endRow*SQ_SIZE))
```

First, create `def highlightSquares` in ChessMain. Get the parameters that are reference to highlighting moves into the function, where game_state is calling `ChessEngine.GameState()` from the Chess Engine.

When we're highlighting a square, we want to make sure that the square selected is not empty meaning that if the user hasn't clicked on anything and we don't need any highlighting. Now set the values r, c as the reference to the rows and columns that the user has selected. Then check whether the square that's been selected is equal to white or black turns by checking the board location.

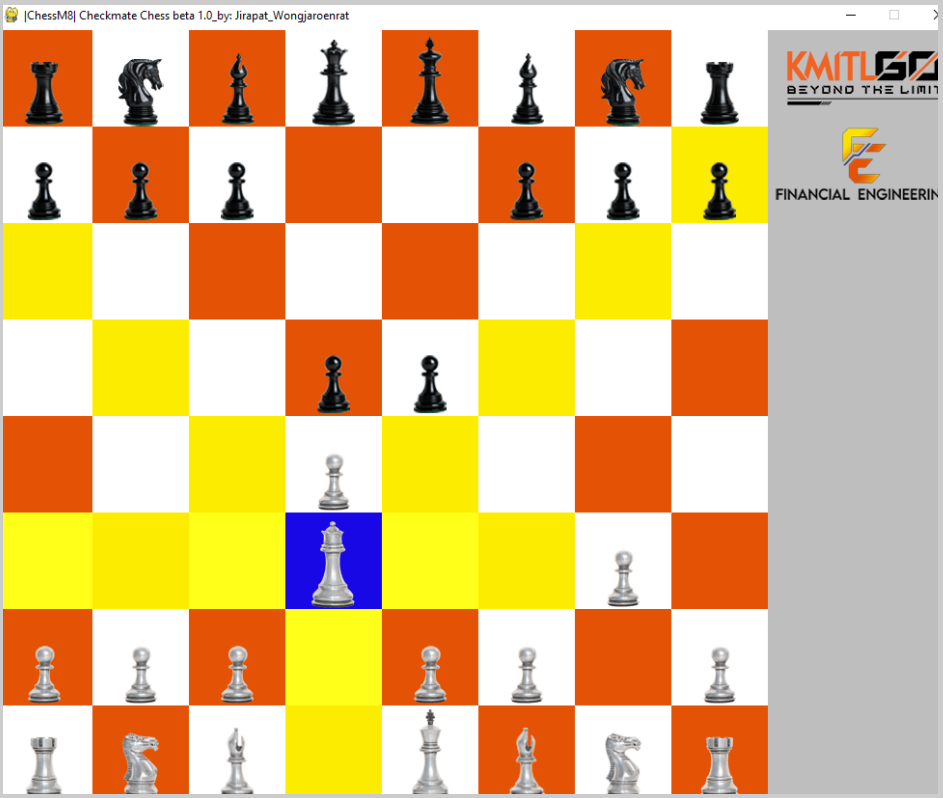
Using transparency feature in pygame `p.Surface` to draw a rectangle above the chess piece coordinates (x, y). `s.set_alpha` sets a transparency value and chooses the color.
Then use our screen variable `screen.blit` which is the pygame surface where everything is being drawn on to draw on top.

For the valid moves of the selected piece. We then go through the valid moves to check if the start row & start column of that move is equal to our selected square. And we can blit our new colored surface.

```
#---- add in chess piece highlighting, move suggestions later -----#

def drawGameState(screen, game_state, validMoves, sqSelected):
    drawBoard(screen)  # draw squares on the board (should be called before drawing anything else)
    highlightSquares(screen, game_state, validMoves, sqSelected)
    drawPieces(screen, game_state.board)  # draw piece on top of the squares
```

Call in the highlightSquare function in `def drawGameState` which is responsible for drawing the images on top of each other. In this case we draw the objects in this order: Chessboard > highlight > Chess Pieces. For the user to see the highlight squares clearly while not blocking the selected piece and the original chess board.

Square highlighting test
https://drive.google.com/file/d/1RoT2b19oe-jZNa3lOYKHHuCDqBLNspyH/view?usp=sharin



- *Chess piece Moves Animation*

_In this idea,when the user clicks to move the piece to the selected square, instead of immediately redrawing the piece again, we can slow this down frame by frame to make the smooth transition from each square.

Starting with the change in rows & columns (delta R, C). Frames per square essentially controls the animation speed, how many fps you want it to take to move 1 square is 10 or we can play with it. `frameCount` is the total amount of frames this animation will take. Basically framesPerSquare times how many square I move which is the absolute value of dR plus dC
In the for loop, for each frame how much does the column and row move by and to find the ratio of how far through the animation I am which is frame/framecount. `move.startRow + dR * frame / frameCount`

For each frame of the animation we're going to draw the board where we're going to draw all the pieces. And we're able to draw a rectangle on the screen at a given color at the end square location.

```
219     def animateMoves(move, screen, board, clock):
220         global colors
221         dR = move.endRow - move.startRow
222         dC = move.endCol - move.startCol
223         framePerSquare = 10  # frames to move 1 square
224         frameCount = (abs(dR) + abs(dC)) * framePerSquare
225         for frame in range(frameCount + 1):
226             r, c = (move.startRow + dR * frame / frameCount, move.startCol + dC * frame / frameCount)
227             drawBoard(screen)
228             drawPieces(screen, board)
229             # erase the piece moved from its ending square
230             color = colors[(move.endRow + move.endCol) % 2]
231             endSquare = p.Rect(move.endCol*SQ_SIZE, move.endRow*SQ_SIZE, SQ_SIZE, SQ_SIZE)
232             p.draw.rect(screen, color, endSquare)
233             # draw captured piece onto rectangle
234             if move.pieceCaptured != "--":
235                 screen.blit(IMAGES[move.pieceCaptured], endSquare)
236             # draw moving piece
237             screen.blit(IMAGES[move.pieceMoved], p.Rect(c*SQ_SIZE, r*SQ_SIZE, SQ_SIZE, SQ_SIZE))
238             p.display.flip()
239             clock.tick(60)
```

Recall the `def animateMoves` in Main function right under when we actually make a move and also set the True/False variable in player click and undo move to tell whether to animate the moves.

```
131             # ------key handlers------ #
132             elif event.type == p.KEYDOWN:
133                 if event.key == p.K_z:  # Undo when "z" is being pressed
134                     game_state.undoMove()
135                     moveMade = True
136                     animate = False
137 
138         if moveMade:
139             if animate:
140                 animateMoves(game_state.moveLog[-1], screen, game_state.board, clock)
141             validMoves = game_state.getValidMoves()
142             moveMade = False
143             animate = False
```

Animate moves test
https://drive.google.com/file/d/1XswZ36tbLUCUS1m30y77vBbSRs6H4UDu/view?usp =sharing

- *UI improvement: Reset game & end-game text*

```
131             # ------key handlers Key Press------ #
132             elif event.type == p.KEYDOWN:
133                 if event.key == p.K_z:  # UNDO when "z" is being pressed
134                     game_state.undoMove()
135                     moveMade = True
136                     animate = False
137                 if event.key == p.K_r: # RESET the board when "R" is pressed
138                     game_state = ChessEngine.GameState()
139                     validMoves = game_state.getValidMoves()
140                     sqSelected = ()
141                     playerClicks = []
142                     moveMade = False
143                     animate = False
```

For the resetting board, it's a straightforward one to make. By adding another key event in the main event under the Undo key that we've made earlier. When pressed the "r" key, this will reset the board. We'll re-initialize the gamestate then we're going to reset the valid moves, and undo any square selected by the user and set our flag variables back to default.

I include this feature since when the game is over or you wanted to restart in the middle of the game without having to re-launch the program. It's also good for debugging since we can always go back to the beginning and test out different things without wasting time.

For Endgame text, I'll let it appear when the game is over on the following conditions: Black or White wins by Checkmate, or a draw due to Stalemate.

```
260 def drawText(screen, text):
261     font = p.font.SysFont("Helvitca", 70, True, True)
262     textObject = font.render(text, 0, p.Color("White"))
263     textLocation = p.Rect(0, 0, WIDTH, HEIGHT).move(WIDTH/2 - textObject.get_width()/2, HEIGHT/2 - textObject.get_height()/2)
264     screen.blit(textObject, textLocation)
265     textObject = font.render(text, 0, p.Color("Black"))
266     screen.blit(textObject, textLocation.move(2, 2))
267 
268     shadow = p.font.SysFont("Arial", 30, True, True)
269     textObject2 = shadow.render("Thanks for playing!!", True, p.Color("Grey"))
270     screen.blit(textObject2, (300, 470))
271     font2 = p.font.SysFont("Arial", 30, True, True)
272     textObject2 = font2.render("Thanks for playing!!", True, p.Color("Red"))
273     screen.blit(textObject2, (302, 472))
```

Starting by creating `def drawText` In pygame, text cannot be written directly to the screen. The first step is to create a Font object with a given font size. The second step is to render the text into an image with a given color. The third step is to blit the image to the screen.

```
154 
155         if game_state.checkmate:
156             gameOver = True
157             if game_state.whiteToMove:
158                 drawText(screen, "Black wins by checkmate")
159             else:
160                 drawText(screen, "White wins by checkmate")
161         elif game_state.stalemate:
162             gameOver = True
163             drawText(screen, "STALEMATE DRAW")
164 
165         clock.tick(MAX_FPS)
166         p.display.flip()
167         p.display.update()
```

We can now call the drawText function in the main event above to check if it was a checkmate or stalemate and send the text back to the function to blit on the screen.

Reset game & Endgame text test
https://drive.google.com/file/d/1KLz3wYql7o9FIZabjY4DreA5ekK375gY/view?usp=sharing

11/12/2021

Chess Algorithm: Creating Chess Bot AI & GameModes

Creating a new ChessBot.py file to calculate RANDOM move from the list of valid moves.

```python
import random


def findRandomMove(validMoves):
    return random.choice(validMoves)

```

Simple Random AI Algorithm (noob bot)

We then set 2 initial variables for Player 1 and Player 2 in our Main function, as we're going to add the game mode: Player vs Player, Player vs Bot, or Bot vs Bot

```python
    # ------- Choosing GAME MODE ---------#
    Player_One_Human = False  # If a Human is playing white, then this will be True. If AI is playing, then this will be False
    Player_Two_Human = True  # Same as above
    # ---------------------------------- #
    if Player_One_Human:
        gameMode(screen, "Player 1 VS Bot")
    elif Player_Two_Human:
        gameMode(screen, "Player 2 VS Bot")
    else:
        gameMode(screen, "Bot VS Bot")

    ############## Game driver : Don't touch ##############
    while running:
        humanTurn = (game_state.whiteToMove and Player_One_Human) or (not game_state.whiteToMove and Player_Two_Human)

        for event in p.event.get():
            if event.type == p.QUIT:  # ---- Event -----#
                running = False
```

Add in conditions to check if which color is on move (line 110), making sure that human is playing as player one or two. Now we can just change those parameters Player_One & Player_Two in order to change game modes

Below the game events in the Main function, we then add this simple AI move finder logic. I'm trying to make the AI move automatically without any click, so basically it'll only check if it's not the human turn then it'll be the AI's turn and they should try to make a move.

We then pass in our validMoves into the findRandomMove function from ChessBot.py that we've implemented before. And set the rest of the parameters like we did earlier.

```python
            # AI Move finder logic
            if not gameOver and not humanTurn:
                AIMove = ChessBot.findRandomMove(validMoves)
                game_state.makeMove(AIMove)
                moveMade = True
                animate = True
```

To choose whether to play a 2 player game or vs Computer or just see Computer Playing against itself, make the following changes in the Config File:

- For Playing as White (Default) : Set Player_One_Human = True and Player_Two_Human = False
- For Playing as Black : Set Player_Two_Human = True and Player_One_Human = False
- For 2 Player Game : Set both Player_Two_Human and Player_Two_Human = True
- For seeing Computer play from both sides : Set both Player_One_Human and Player_Two_Human = False

```
# ------- Choosing GAME MODE ---------#
Player_One_Human = False  # If a Human is
Player_Two_Human = True  # Same as above
```

Chess AI & different GameModes test
https://drive.google.com/file/d/1hxNtYNXpZSGXGQrKPzHy-33zbrLjkIwY/view?usp=sharing

12/12/2021

Chess Bot Advanced-Algorithm: Making Chess AI Smarter with Greedy/MinMax Algorithm

In this section, we're going to make our Chess AI to think more logically like humans by inputting an algorithm onto it, instead of making it randomly move. In chess, we need to be looking at more moves in advance as a main strategy to gain advantage and win the game.

Greedy Algorithm is where the AI will look at all of the possible responses to the last player turns moves. It'll then evaluate the best possible result of that opponent's move to pick the best move. For example, should it capture the opponent piece or to trade-in piece captured, etc.

In other words, the Greedy Algorithm is that we're trying to maximize our own scores. But how do we know when one side is winning on the other and what's the good board position?

In human perspective, if we look at the material alone each side starts with 8 pawns, 2 rooks, 2 bishops, 2 knights, 1 king and 1 queen. When the pieces were exchanged, some pieces are considered to be more powerful than the others. We can assume that the Bishop and Knight are viewed to be about the same, the Queen is more powerful than a Rook. The Rook is considered to be more powerful than both Bishop & Knight. And the Pawn is the least powerful piece.

```
# A map of piece to score value -> Standard chess scores
pieceScore = {"K": 0, "p": 1, "N": 3, "B": 3, "R": 5, "Q": 9} # making King = 0, as no one can capture the K
CHECKMATE = 1000
STALEMATE = 0
```

So we're going to add the scoreboard system and value each chess piece with scores.

```
26  def scoreMaterial(board):
27      score = 0
28      for row in board:
29          for square in row:
30              if square[0] == "w":
31                  score += pieceScore[square[1]]
32              elif square[0] == "b":
33                  score -= pieceScore[square[1]]
34      return score
```

Gives the score of the board according to the material on it . White piece positive material and Black piece negative material. Assuming that Human is playing White and BOT is playing black

First, we create a for loop that goes through our board. Then we'll check if there's a piece in that square and if it's a white or black piece which will score based on our dictionary

A perfectly even game will equal to zero, if white has the material advantage and the score will be positive and if the black has advantage the score will be negative. So we access our `pieceScore` based on the character at `square[1]` which could be a pawn or well add that value later.

After we have the score, now we can look for the best move.

13/12/21

Chess Bot Advance-Algorithm: Implementing Min/Max algorithm to AI (continued)

In the section above, we've already generated an AI algorithm to value each of the chess pieces. Now we then create a function to find the best move based on the list of valid moves. I've set the initial score for the AI (playerMaxScore) as -CHECKMATE, which is equal to -1000. as AI is playing Black this is the worst possible score. AI will start from worst and try to improve

With the Min/Max algorithm, we can use these for loops to try to minimize the opponent's score based on the score values.
By the way, the algorithm will be much more complicated as we go deeper. And soon it will be too hard to debug. So I decided to divide it into different functions and import it into Main.

```
def findBestMove(game_state, validMoves):
    turnMultiplier = 1 if game_state.whiteToMove else -1 # for allowing AI to play as any color
    playerMaxScore = -CHECKMATE
    bestMove = None
    random.shuffle(validMoves)
    for playerMove in validMoves:
        game_state.makeMove(playerMove)
        opponentMinScore = CHECKMATE
        opponentMoves = game_state.getValidMoves()
        if game_state.checkmate:
            game_state.undoMove()
            return playerMove
        elif game_state.stalemate:
            opponentMinScore = STALEMATE
        else:
            for opponentMove in opponentMoves:
                game_state.makeMove(opponentMove)
                game_state.getValidMoves()
                if game_state.checkmate:
                    score = -CHECKMATE
                elif game_state.stalemate:
                    score = STALEMATE
                else:
                    score = turnMultiplier * scoreMaterial(game_state.board)
                if score < opponentMinScore:
                    opponentMinScore = score
                game_state.undoMove()
        if playerMaxScore < opponentMinScore:
            playerMaxScore = opponentMinScore
            bestMove = playerMove
        game_state.undoMove()
        return bestMove
```

I'll be using `def findBestMove` As a primary algorithm for the AI, in which it'll evaluate the best move. But there's a scenario that you'll have to trade in or do a random move because there's no possible moves left.

So if there's no moves available, we'll set the AI to move randomly without thinking like usual.

```
# AI Move finder logic
if not gameOver and not humanTurn:
    AIMove = ChessBot.findBestMove(game_state, validMoves)
    if AIMove is None:
        AIMove = ChessBot.findRandomMove(validMoves)
    game_state.makeMove(AIMove)
    moveMade = True
    animate = True
```

- Implementing MinMax Recursively

```python
def findBestMoveMinMax(game_state, validMoves):
    global nextMove
    nextMove = None
    findMoveMinMax(game_state, validMoves, DEPTH, game_state.whiteToMove)
    return nextMove


def findMoveMinMax(game_state, validMoves, depth, whiteToMove):
    global nextMove
    if depth == 0:
        return scoreMaterial(game_state.board)

    global nextMove
    if game_state.whiteToMove:  # Try to maximize score
        maxScore = -CHECKMATE
        for move in validMoves:
            game_state.makeMove(move)
            nextMove = game_state.getValidMoves()
            score = findMoveMinMax(game_state, nextMove, depth-1, False)
            if score > maxScore:
                maxScore = score
                if depth == DEPTH:
                    nextMove = move
            game_state.undoMove()
        return maxScore

    else:  # Try to minimize score
        minScore = CHECKMATE
        for move in validMoves:
            game_state.makeMove(move)
            nextMove = game_state.getValidMoves()
            score = findMoveMinMax(game_state, nextMove, depth-1, True)
            if score < minScore:
                minScore = score
                if depth == DEPTH:
                    nextMove = move
            game_state.undoMove()
        return minScore
```

This is another Min/Max algorithm method (recursive) to make our AI think ahead much more in depth.

Compared to `def findBestMove` is quite similar. Making a move getting our opponents move scoring that board finding the min-max can go only in depth 1, but with this method. The algorithm can now look as many moves ahead as we want since we've set the depth parameters

```python
DEPTH = 2  # Depth for recursive calls
```

With this algorithm, all we have to do is change the value of “DEPTH”, e.g, 3. Now the algorithm will now look 3 moves ahead. So we have more control over how much we want our computer to think

14/12/21

Chess Bot Advance-Algorithm: Implementing Nega Max and Alpha Beta Pruning Algorithm for AI

In this section, I'm going to implement the algorithm called NegaMax, which is a combination of the Min/Max method and then also an Alpha Beta Pruning.

The Min/Max algorithm doesn't provide that efficiency. One of the problems is that the min max is checking every single position regardless of how it scores. What Alpha Beta Pruning will allow us to do is basically to not visit branches that we know which won't be any better than our best possible score. So it allows us to cut off branches of our game state tree that we don't need to visit, since our opponent will play optimally. So there's no point in visiting some of the move branches which will take an AI to think longer.

```python
"""
Best Move Calculator using NegaMax Algorithm along with Alpha Beta Pruning
"""


def findMoveNegaMaxAlphaBeta(game_state, validMoves, depth, alpha, beta, turnMultiplier):
    global nextMove
    if depth == 0:
        return turnMultiplier * scoreBoard(game_state)
    # Move ordering - implement later
    maxScore = -CHECKMATE
    for move in validMoves:
        game_state.makeMove(move)
        nextMoves = game_state.getValidMoves()
        score = -findMoveNegaMaxAlphaBeta(game_state, nextMoves, depth - 1, -beta, -alpha, -turnMultipl
        if score > maxScore:
            maxScore = score
            if depth == DEPTH:
                nextMove = move
        game_state.undoMove()
        if maxScore > alpha:  # pruning happens
            alpha = maxScore
        if alpha >= beta:
            break
    return maxScore
```

```python
def findMoveNegaMax(game_state, validMoves, depth, turnMultiplier):
    global nextMove
    if depth == 0:
        return turnMultiplier * scoreBoard(game_state)

    maxScore = -CHECKMATE
    for move in validMoves:
        game_state.makeMove(move)
        nextMoves = game_state.getValidMoves()
        score = -findMoveNegaMax(game_state, nextMoves, depth - 1, -turnMultiplier)
        if score > maxScore:
            maxScore = score
            if depth == DEPTH:
                nextMove = move
        game_state.undoMove()
    return maxScore
```

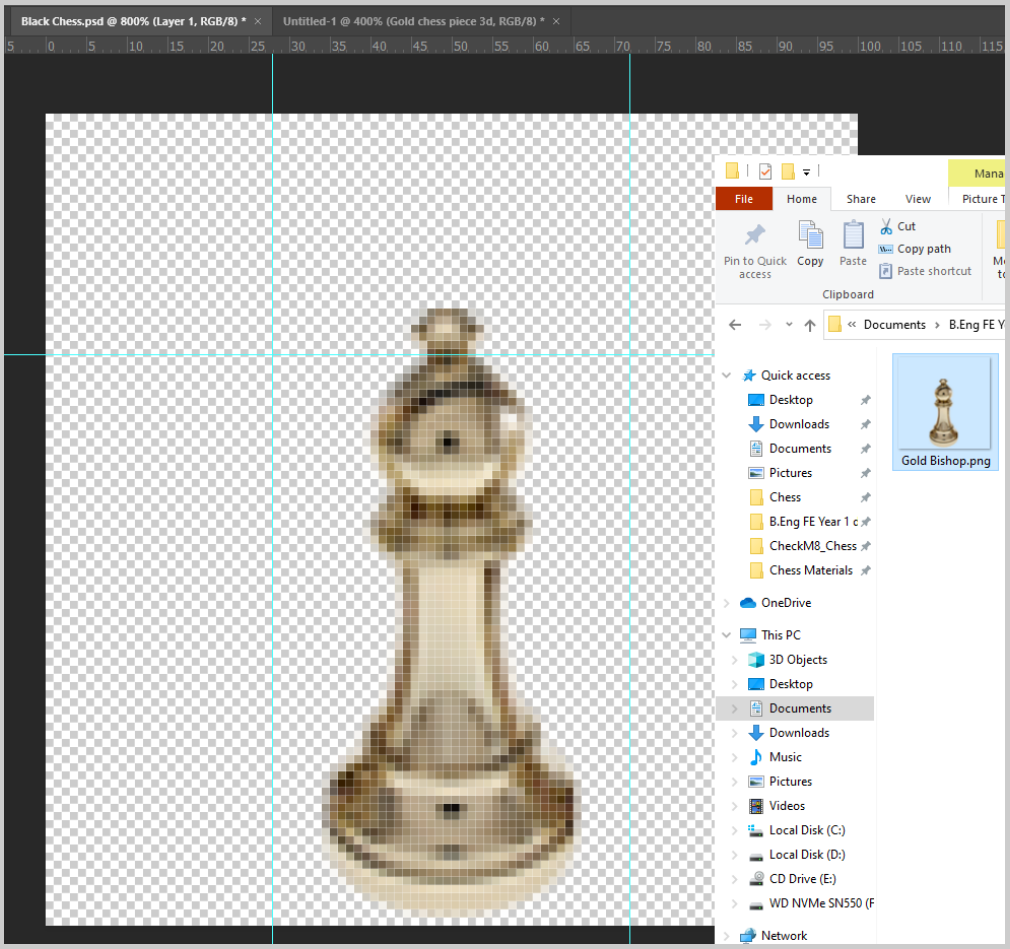
This function helps to call the recursion for the 1st time

```
def findBestMoveMinMax(game_state, validMoves):
    global nextMove
    nextMove = None
    random.shuffle(validMoves)
    findMoveNegaMaxAlphaBeta(game_state, validMoves, DEPTH, -CHECKMATE, CHECKMATE, 1 if game_state.white
    # findMoveNegaMax(gs, validMoves, DEPTH, 1 if gs.whiteToMove else -1)     # For using Nega Max Algor
    # findMoveMinMax(gs, validMoves, DEPTH, gs.whiteToMove)     # For using Min-Max Algorithm
    return nextMove
```

Currently, we've 3 methods to implement the AI algorithm. We can change the algorithm type here.

- **<u>The making of additional piece colors : Gold chess pieces</u>**

Just like the rest of the pieces, the height has to be consistent, to make sure it fits perfectly on each square on the board while remaining high-resolution.



bB - black Bishop	75px
bK - black King	93px
bN - black Knight	70px
bp - black pawn	60px
bQ - black Queen	90px
bR - black Rook	65px

###pawn - height (60 px)

###rest - height << (100px)

Smarter AI & Gold piece test
https://drive.google.com/file/d/1u2ftq7DAkPhhXObBz5cFgAhIg0UVui4o/view?usp=sharing

14/12/21

Chess Main Ui: Creating a Move Log Display

Move Log Display is a feature where users can see the record of each move without looking back at the IDE terminal, but now directly on the UI. We can make this happen by drawing a rect fill on the side of the screen and carry out `self.moveLog` and plug it in our `def drawMoveLog.` Then we can now call this function in `def drawGameState` Function to be drawn over the screen.

```
278  def drawMoveLog(game_state):
279      moveLogRect = p.Rect(WIDTH, 200, 260, 550)
280      p.draw.rect(screen, p.Color("brown"), moveLogRect)
281      moveLog = game_state.moveLog
282      moveTexts = []
283      for i in range(0, len(moveLog), 2):
284          moveString = str(i//2 + 1) + "." + moveLog[i].GetChessNotation() + " "
285          if i+1 < len(moveLog): #make sure black made a move
286              moveString += moveLog[i+1].GetChessNotation() + "__"
287          moveTexts.append(moveString)
288
289      movesPerRow = 3
290      horizontalPadding = 5
291      lineSpacing = 7
292      verticalPadding = 5
293      for i in range(0, len(moveTexts), movesPerRow):
294          text = ""
295          for j in range(movesPerRow):
296              if i + j < len(moveTexts):
297                  text += moveTexts[i+j]
298          textObject = moveLogFont.render(text, True, p.Color("white"))
299          textLocation = moveLogRect.move(horizontalPadding, verticalPadding)
300          screen.blit(textObject, textLocation)
301          verticalPadding += textObject.get_height() + lineSpacing
```

```
def drawGameState(game_state, validMoves, sqSelected):
    drawBoard()  # draw squares on the board (should be called before drawing anything else)
    highlightSquares(game_state, validMoves, sqSelected)
    drawPieces(game_state.board)  # draw piece on top of the squares
    drawMoveLog(game_state)
```

Move Log Display Test
https://drive.google.com/file/d/1S2MZJg5UBEDKCnesgr7a68Mu3UZmxq8f/view?usp=sharing

15/12/21

Final AI Improvement with threads

How AI methods work:

1. We look at our piece scores and move the piece based on def findbestmove. It's responsibility is to call the initial recursive call to the algorithm methods and return the results at the end.

2. We can compare each AI algorithm method efficiency by implementing a counter to check when each time the algorithm is being called in each move and print out the value. The larger the number, the deeper it'll think to move each piece and therefore takes more time.

```
61  def findBestMove(game_state, validMoves):
62      global nextMove, counter
63      nextMove = None
64      random.shuffle(validMoves)
65      counter = 0
66      # findMoveNegaMaxAlphaBeta(game_state, validMoves, DEPTH, -CHECKMATE, CHECKMATE, 1 it
67      findMoveNegaMax(game_state, validMoves, DEPTH, 1 if game_state.whiteToMove else -1)
68      # findMoveMinMax(game_state, validMoves, DEPTH, game_state.whiteToMove)     # For usi
69
70      print(counter)
71      return nextMove
```

```
Hello from the pygame comm
60
61
61
130
80
95
61
CHECK! Player 1
```

From the counter parameter, you can see `def findMoveNegaMax` method with depth = 2 has printed out the number of the execute value as shown when the AI is evaluating the moves. These moves it made are considered as fast but easy.

```
61  def findBestMove(game_state, validMoves):
62      global nextMove, counter
63      nextMove = None
64      random.shuffle(validMoves)
65      counter = 0
66      findMoveNegaMaxAlphaBeta(game_state, validMoves, DEPTH, -CHECKMATE, CHECKMATE, 1 if game_state.whit
67      # findMoveNegaMax(game_state, validMoves, DEPTH, 1 if game_state.whiteToMove else -1)     # For usi
68      # findMoveMinMax(game_state, validMoves, DEPTH, game_state.whiteToMove)     # For using Min-Max Alg
69
70      print(counter)
71      return nextMove
```

```
Hello from the pygame comm
586
578
801
1323
2500
1229
2260
1114
```

Now compare with `def findMoveNegaMaxAlphaBeta` method with depth = 3 has printed out the value which is a lot greater than theNegaMax. These moves it made are considered to be more complex as we increase the depth, making the AI more smarter but takes a long time to evaluate and may lead to not responding to program or user game crashes.

- ChessBot Threading (additional Ai algorithm)

There are many different ways to implement positional type play. One thing we could do is we could look at the list of moves that we currently have available to us.
So we could count how many attacking moves we have. Generally speaking, the more attacking moves you have, the better your position. Therefore the more threads we can make.

In this part, I'm going to create threads in our AI by saying that the more attacks you can make from a position or the more captures you can make the better off you are.
Starting by simply looking at the position of pieces in our board and defining positional type arrays. We can create a 2-dimensional map of where we want our pieces to be positioned (Scale 1-4). And rate each piece position that can threaten other squares in scale 1-4.

```
3   # A map of piece to score value -> Standard chess scores
4   pieceScore = {"K": 0, "p": 1, "N": 3, "B": 3, "R": 5, "Q": 10}  # making King = 0, as no one can captur
5
6   knightScores = [[1, 1, 1, 1, 1, 1, 1, 1],
7                   [1, 2, 2, 2, 2, 2, 2, 1]
8                   [1, 2, 3, 3, 3, 3, 2, 1]
9                   [1, 2, 3, 4, 4, 3, 2, 1]   # The Knight at the center of the board is better than being
10                  [1, 2, 3, 4, 4, 3, 2, 1]   # outside of the board, since it can reach more squares.
11                  [1, 2, 3, 3, 3, 3, 2, 1]   # AI will consider the higher value as a better move. Because
12                  [1, 2, 2, 2, 2, 2, 2, 1]
13                  [1, 1, 1, 1, 1, 1, 1, 1]]
14
15  piecePositionScores = {"N": knightScores}
```

Then we create a dictionary called piecePositionScores to map each of my pieces to the appropriate 2-dimensional array.

```
224 def scoreBoard(game_state):
225     if game_state.checkmate:
226         if game_state.whiteToMove:
227             return -CHECKMATE   # BLACK WINS
228         else:
229             return CHECKMATE
230     if game_state.stalemate:
231         return STALEMATE
232
233     score = 0
234     for row in game_state.board:
235         for square in row:
236             if square[0] == 'w':
237                 score += pieceScore[square[1]]
238             elif square[0] == 'b':
239                 score -= pieceScore[square[1]]
240     return score
```

For the Knight, we're going to map them to the Knight scores array. The AI will now look up the row can column in this table and evaluate this.
We now implement this in our score method.

We've already gone through the rows & columns and added the pieces in.
"If it's white or black then we added in a certain score. (line 209-217).

Now we have to modify this for loop to be able to add the piecePositions in.

```
199     score = 0
200     for row in range(len(game_state.board)):  # len = 8
201         for col in range(len(game_state.board[row])):
202             square = game_state.board[row][col]
203             if square != "--":
204                 # score it positionally based on what type of piece it is
205                 if square[1] == "N":
206                     piecePositionScore = piecePositionScores["N"][row][col]
207
208                 if square[0] == "w":
209                     score += pieceScore[square[1]] + piecePositionScore
210                 elif square[0] == "b":
211                     score -= pieceScore[square[1]] + piecePositionScore
212
213     return score
```

What this does is it looks up in the table `piecePositionScores = {"N": knightScores}`
And it returns the KnightScores array.
Now we have to find the row & column value in this array and get that numbers
To weight our Ai.

Now we'll look to see if it's a white or black piece and our score will be the material of the piece + `piecePositionScores`.

If it's a black piece, which is minus equal the score. Then we're going to minus the sum of the score of the material of the piece + `piecePositionScores`. So what we should see now is that our Ai has the tendency to move.

```python
def scoreBoard(game_state):
    if game_state.checkmate:
        if game_state.whiteToMove:
            return -CHECKMATE  # Black wins
        else:
            return CHECKMATE  # White wins
    if game_state.stalemate:
        return STALEMATE

    score = 0
    for row in range(len(game_state.board)):  # len = 8
        for col in range(len(game_state.board[row])):
            square = game_state.board[row][col]
            if square != "--":
                # score it positionally based on what type of piece it is
                piecePositionScore = 0
                if square[1] != "K": # no position table for king
                    if square[1] == "p":  # for pawns
                        piecePositionScore = piecePositionScores[square][row][col]
                    else: #for other pieces
                        piecePositionScore = piecePositionScores[square[1]][row][col]

                if square[0] == "w":
                    score += pieceScore[square[1]] + piecePositionScore * 0.1 # 0.1 to
                elif square[0] == "b":
                    score -= pieceScore[square[1]] + piecePositionScore * 0.1

    return score
```

```python
bishopScores = [[4, 3, 2, 1, 1, 2, 3, 4]
                [3, 4, 3, 2, 2, 3, 4, 3]
                [2, 3, 4, 3, 3, 4, 3, 2]
                [1, 2, 3, 4, 4, 3, 2, 1]
                [1, 2, 3, 4, 4, 3, 2, 1]
                [2, 3, 4, 3, 3, 4, 3, 2]
                [3, 4, 3, 2, 2, 3, 4, 3]
                [4, 3, 2, 1, 1, 2, 3, 4]]

queenScores = [[1, 1, 1, 3, 1, 1, 1, 1],
               [1, 2, 3, 3, 3, 1, 1, 1],
               [1, 4, 3, 3, 3, 4, 2, 1],
               [1, 2, 3, 3, 3, 2, 2, 1],
               [1, 2, 3, 3, 3, 2, 2, 1],
               [1, 4, 3, 3, 3, 4, 2, 1],
               [1, 1, 2, 3, 3, 1, 1, 1],
               [1, 1, 1, 3, 1, 1, 1, 1]]

rookScores = [[4, 3, 4, 4, 4, 4, 3, 4],
              [4, 4, 4, 4, 4, 4, 4, 4],
              [1, 1, 2, 3, 3, 2, 1, 1],
              [1, 2, 3, 4, 4, 3, 2, 1],
              [1, 2, 3, 4, 4, 3, 2, 1],
              [1, 1, 2, 3, 3, 2, 1, 1],
              [4, 4, 4, 4, 4, 4, 4, 4],
              [4, 3, 4, 4, 4, 4, 3, 4]]
```

```python
piecePositionScores = {"N": knightScores, "Q": queenScores, "B": bishopScores,
                       "R": rookScores, "wp": whitePawnScores, "bp": blackPawnScores}
```

Now, our Ai algorithm is successfully done, it'll play a little bit smarter than it was before but with the downside of slow speed.

Ai Algorithms methods:

1. `def findRandomMove` Function to calculate Random move from the list of valid moves.

2. `def findBestMove` Helper method to call recursion for the first time in Main.

3. `def findMoveMinMax` An Ai algorithm method 1, to find the best move based on material itself (call-in def findBestMove)

4. `def findMoveNegaMax` An Ai algorithm method 2, move calculator using NegaMax Algorithm.

5. `def findMoveNegaMaxAlphaBeta` An Ai algorithm method 3, BEST Move calculator using NegaMax Algorithm along with Alpha Beta Pruning.
