 In Association with 



CHESS GAME USING ARTIFICIAL INTELLIGENCE (AI) & PERIPHERAL DEVICES

**INTERNSHIP TRAINING - PROJECT REPORT**

**Submitted by**

**J.K.K.NATTRAJA COLLEGE OF ENGINEERING & TECHNOLOGY,**

**KOMARAPALAYAM,**

**NAMAKKAL-638183.**

**Submitted on**

**09-10-2021**

|  |  |  |
| --- | --- | --- |
|  | **CONTENTS** |  |
|  | **ABSTRACT** | **4** |
| **1.** | **INTRODUCTION**  **1.1 Purpose**  **1.2 Scope**  **1.3 Objectives**  **1.4 Abbreviations**  **1.5 Definition**  **1.6 Time** | **5-8** |
| **2.** | **OVERALL DESCRIPTION**  **2.1 Product Perspective**  **2.2 Product Functions**  **2.3 User Characteristics**  **2.4 Operating Environment** | **9** |
| **3.** | **USER REQUIREMENTS**  **3.1 Player vs AI**  **3.2 player1 vs player2**  **3.3 Control by mobile**  **3.4 Control by voice** | **10** |
| **4.** | **FUNCTIONAL REQUIREMENTS**  **4.1.Abbreviations**  **4.2.System requirements** | **11-13** |
| **5.** | **USE CASES**  **5.1 Connect**  **5.2 Move**  **5.3 Flowchart**  **5.4 Pseudocode** | **14-80** |
| **6.** | **EXTERNAL INTERFACE REQUIREMENTS**  **6.1 User interface** | **81-97** |
| 7. | **LIMITATIONS** | **98** |
| **8.** | **SCOPE FOR DEVELOPMENT** | **99** |
| **9.** | **SOURCE CODE** | **100-150** |
| **10.** | **CONCLUSION** | **151** |
| **11.** | **TEAM MEMBERS** | **152** |

**ABSTRACT**

In recent years, computer games have become a common form of entertainment. Fast advancement in computer technology and internet speed have helped entertainment software developers to create graphical games that keep a variety of players’ interest. The emergence of artificial intelligence systems have evolved computer gaming technology in new and profound ways.

In many cases, computer abilities must be toned down in order to give the human player a competitive chance in the game. This improves the human player’s perception of fair game play and allows for continued interest in playing. A proper adaptive learning mechanism is required to further this human player’s motivation. During this project, past achievements of adaptive learning on computer chess game play are reviewed and adaptive learning mechanisms in computer chess game play are proposed. Adaptive learning is used to adapt the game’s difficulty level to the players’ skill levels. This adaptation is done using the player’s game history and current performance. The adaptive chess game is implemented through the open source chess game engine Beowulf, which is freely available for download on the internet.

# **INTRODUCTION**

The aim of the project is to provide both chess in one application such that a single player against AI or two players play against each other.

## 1.1 Purpose

The purpose of this document is to provide a detailed requirements overview for the chess game application. Chess application shall provide the user with a reliable, well designed chess game. This SRS shall cover the full scope of chess applications.

## 1.2 Scope

* That allows two players to engage in a real-time Chess game controlled by voice and mobile application. The primary Goal is to provide a chess game that is intuitive and entertaining for player of all skill levels.
* Chess game is developed entirely in python and can be launched on a system that supports the IDLE.
* This document will be used by the end-users, tester, and developers of the game.

## 1.3 Objectives

The objectives of this project are:

a) To provide a user-friendly interactive environment to the users of the application that helps them to play and communicate with a lot of ease.

b) To provide help to the users in playing the chess that is the different moves of the different pieces are being explained to the users, if they require.

## 1.4Abbreviations

**Term Definition**

HMC Half Move Clock

ENP En-Passant

## 1.5 Definitions

**Chess :** It is a type of game named Chaturanga. A game played by 2 people on a chessboard with 16 pieces each.

**Chess Board :** The size of the board is 640x640 pixels. Chess board comprises 64 squares arranged by eight rows and eight columns. Each and every Single Square has the size of 80x80 pixels. The board of the color is black and white as is the Traditional Staunton Chess Board.

**Pieces :** The Black pieces are arranged in the top of the board and the White pieces are arranged in the bottom of the board in two players by default. In single player mode we can select color then it will be arranged at the bottom of the board.

**Movements :** In this chess game the movements of the pieces are based on algebraic notations.

**Bishop** : one of two pieces are the same color that may be moved any number squares diagonally, as long as no other piece blocks its way. One piece always remains on White squares and the other always on Black.

**Castling** : To move the king two squares horizontally and bring the appropriate rook to the square the king has passed over.

**Check :** To make a move that puts the opponent's King under direct attack.

**Checkmate** : a situation in which an opponent’s king is in check and it cannot avoid being captured. This then brings the game to a victorious result.

**En-Passant** : a method by which a pawn that is moved two squares can be captured by an opponent's pawn commanding the square that was passed

**King** : The main piece of the game, checkmating this piece is the object of the game. It can move 1 space in any direction.

**Knight** : This piece can move 1 space vertically and 2 spaces horizontally or 2 spaces vertically and 1 space horizontally. This piece looks like a horse. This piece can also jump over other pieces.

**Pawn** : One of eight men of one color and of the lowest value usually moved one square at a time vertically and captured diagonally. The first move of the pawn can be one or two moves.

**Queen** : This piece can move in any number of spaces in any direction as long as no other piece is in its way.

**Rook** : one of two pieces of the same color that may be moved any number squares horizontally or vertically, as long as no other piece blocks its way.

**Stalemate** : A situation in which a player’s king is not in check, but that player can make no move. This then results in a stalemate, which is a draw.

**HMC :** The Half Move Clock inside a chess position object takes care of enforcing the fifty-move rule. This counter is reset after captures or pawn moves, and incremented otherwise.

**Select Piece** : The selected piece square is Identified by the dark green color.

**Possible ways** : the selected piece movements of possible ways are shown in the light green color. It is a dominant color in nature that makes you think of growth.

**King wins** : if the king wins it can be identified by the Green circle within the position of king.

**King lost**: If the king lost it can be identified by the Red circle within that square box.

**Possible Attack** : If any of the pieces attack other pieces it shows by the red color over the pieces.

**Match Draw** : it can be identified by the Yellow circle of both the Kings square box.

**Previous move :** previous moves are Identified by the Grey color square box.

## 1.6Time

 There is no time limit to a player's turn.

# **2. OVERALL DESCRIPTIONS**

## 2.1 Product Perspective

There is no shortage of computer-based chess games. Chess application is not significantly different from these other games.

The game allows single player with AI of a game either online or offline. It provides a simple GUI. It also allows players to choose the select mode, difficulty level, select control (mobile/voice).

The game allows two players to play the game online or offline. it also allows players to choose only select control.

## 2.2 Product Functions

The following options shall be available in a starting menu:

Single player – Start a game of chess against an AI: Color, Difficulty, Control selection shall be available here

Two-Player – Start a chess game between two players: Control selection only available here.

Exit - If the game finished or not finished, the user can close the window for exit.

## 2.3 User Characteristics

The users of CHESS need experience and the ability to play chess at least at a basic level.

# **3. USER REQUIREMENTS**

## 3.1 Player vs AI:

First the user can select the menu player Vs AI

3.2 Player1 vs Player2:

First the user can select the menu player1 Vs player2

## 3.3. Control by mobile

First the user can select the menu control by mobile

## 3.4. Control by voice

First the user can select the menu control by voice

# 4.FUNCTIONAL REQUIREMENTS

## 4.1 Abbreviations

**Term Definition**

AI Artificial Intelligence

GUI Graphical User Interface

PY Python

IDLE Integrated Development Learning

Environment

GPU Graphics Processing Unit

## 4.2.System requirements:

## 4.2.1 Software requirements:

* Android version 8 and above
* PC remote controller(version 7.4.3) for android and pc
* Internet Connection
* A Chess board and its layout & design
* It supports python IDE (Pyscripter, Pycharm, IDLE)
* It supports various platforms like windows(version 7 and above), linux, mac., etc.,

### 4.2.2 Hardware requirements:

#### 4.2.2.1 PC:

* System Processor : intel core i
* Motherboard : Genuine Intel
* RAM : 4 GB or higher
* Memory : 512 GB or higher
* CPU : i3,i5,i7

4.2.2.2. MOBILE:

* RAM: 2GB and above
* Memory: 16 GB and above

#### 4.2.2.3. REMOTE PC:

This is one of the controller by PC. This application is only working in network connection through pc and mobile same wi-fi connection or USB cable. It’s connected by which hostname of the pc is connected, then it is only controlled to your pc. Finally, we will control the pc on the mobile.

Access your remote computer from any PC, Mac, or Linux machine, IOS, and Android devices, or via the web.

Perform functions like lock screen, blank host screen, adjust the screen of the remote.

This pc remote app has been developed by Monect,Inc.

#### 4.2.2.4. COMMUNICATION INTERFACES:

* Microphone (Headphone version 3.5 mm Jack, system mic)
* GPU : RTX3080

#### 4.2.2.5 Operating Environment

This shall be a python-based application, so the user must have any python Interpreter. Ex: Pyscripter, Pycharm, IDLE.

CHESS is not platform dependent and can be installed in any operating system capable of running any environment.

#### 4.2.2.6. Network:

We connect to the internet via Wi-fi or a nearby connection via or wired connection via USB cable.

# 5.USE CASES

# 5.1. Connect

(i). Users have to select the menu page(Player vs AI & Player1 vs Player2), select color page(black or white), select difficulty(easy, medium, hard), select mode(mobile, voice).

(ii). The chess game always begins with initial moves white pieces only.

Sub flows:

I. User opens the program and runs it in any one of the python Interpreters.

II. Once there is a successful selection(3 modes) a game is started.

III. White pieces players will start the move.

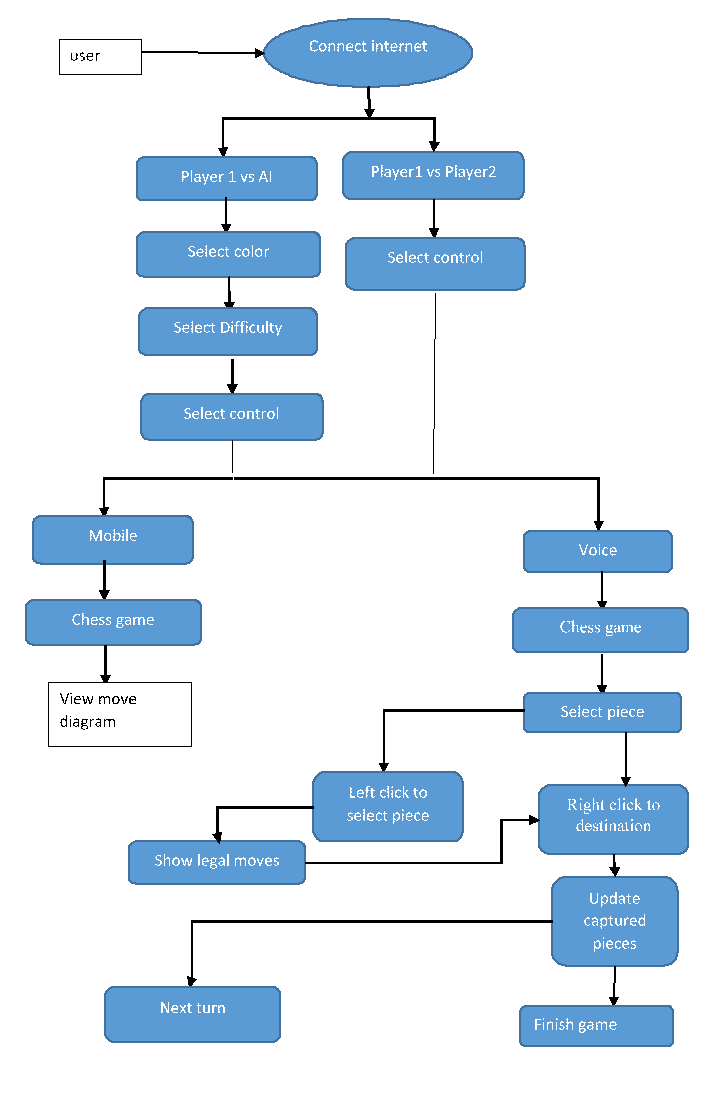


Fig.5.1 .Use case diagram

## 5.2 Move

Precondition: During the Play Game state

Main Flow: The active player clicks a piece to select it. The game displays the positions it can move to.

The player selects the new destination by clicking. The piece is moved there if it is a valid move. Their opponent becomes the active player.

Alternate Flow:

The active player may decide to select a different piece by clicking one of their own.

If there are no valid moves and the active player is not in check the game ends as a stalemate.

If there are no valid moves and the active player is in check the inactive player wins.

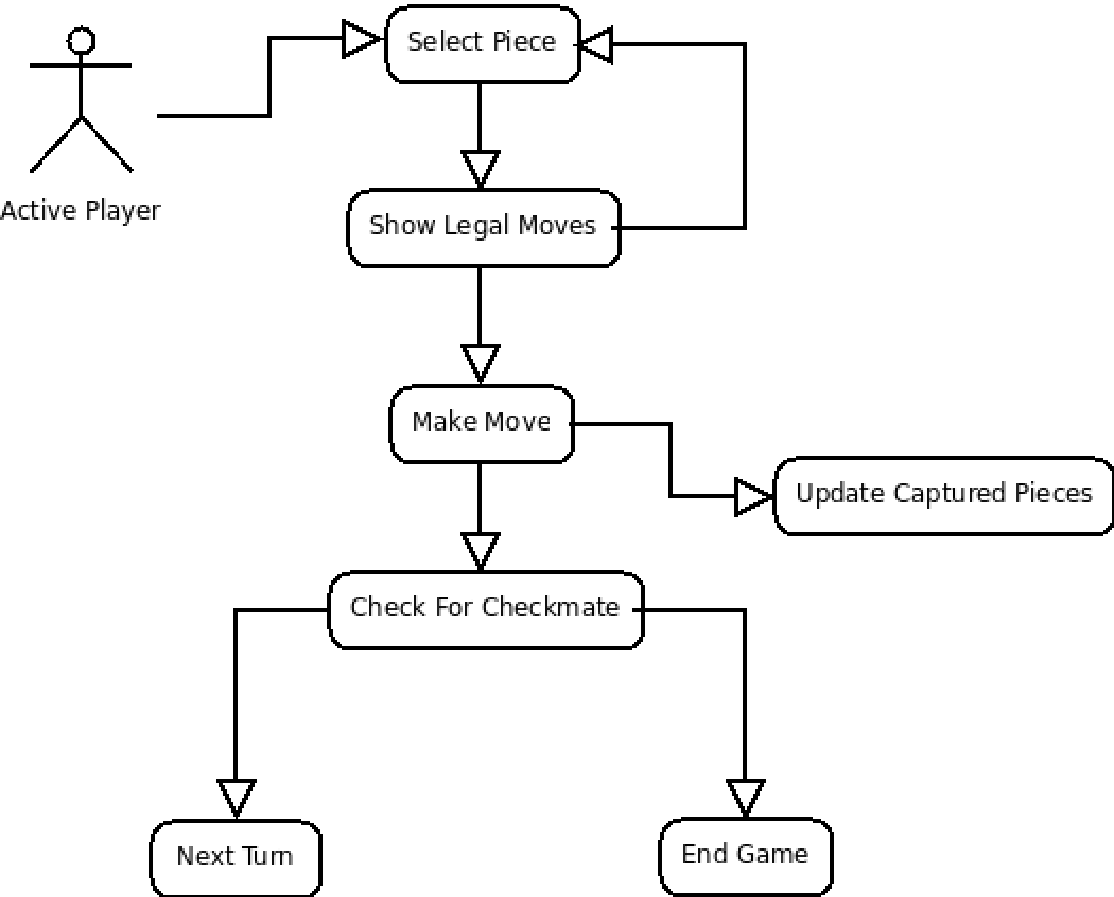


Fig.5.2.Use case diagram for move

## 5.3 Flowchart

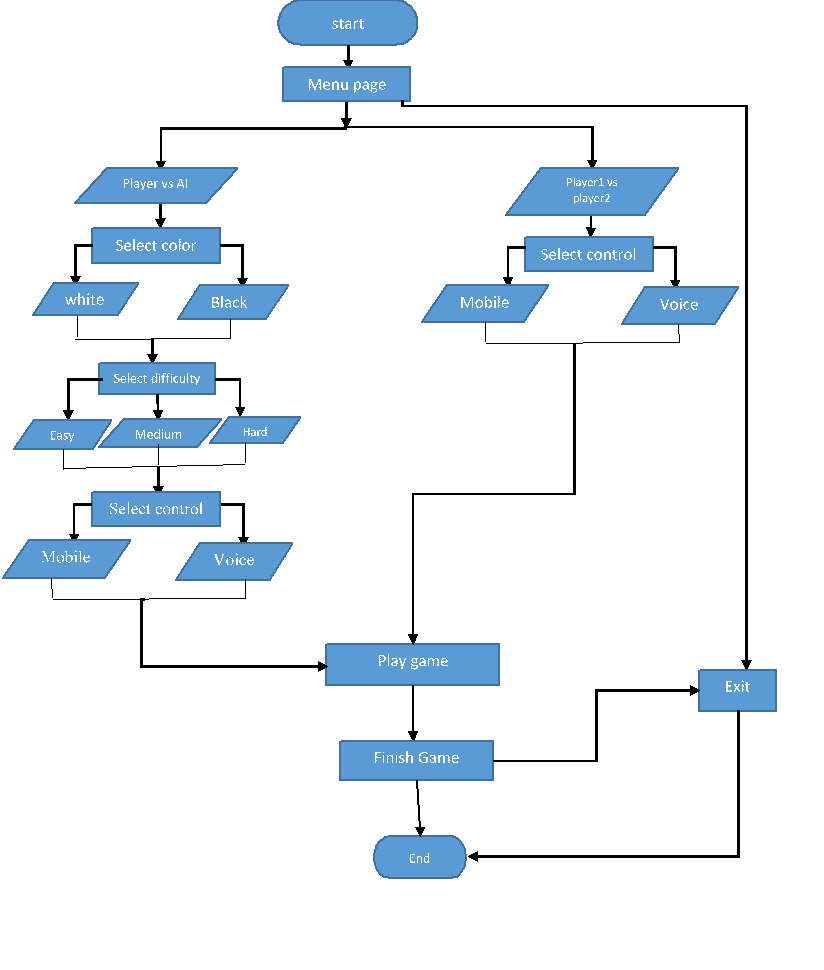


Fig 5.3. Flow chart for controls

# 

FIG.5.4.Flow chart for game

## 5.4 PSEUDOCODE:

## BEGIN

CLASS GAMEPOSITION:

DEF \_\_init\_\_(SELF,BOARD,PLAYER,castling\_rights,EnP\_Target,HMC,history = {}):

SELF.c=Commands()

SELF.BOARD = BOARD

SELF.PLAYER = PLAYER

SELF.EnP = EnP\_Target

SELF.castling = castling\_rights

SELF.HMC = HMC

SELF.history = history

DEF getBOARD(SELF):

RETURN SELF.BOARD

DEF setBOARD(SELF,BOARD):

SELF.BOARD = BOARD

DEF getPLAYER(SELF):

RETURN SELF.PLAYER

DEF setPLAYER(SELF,PLAYER):

SELF.PLAYER = PLAYER

DEF getCastleRights(SELF):

RETURN

SELF.castling

DEF setCastleRights(SELF,castling\_rights):

SELF.castling = castling\_rights

DEF getEnP(SELF):

RETURN SELF.EnP

DEF setEnP(SELF, EnP\_Target):

SELF.EnP = EnP\_Target

DEF getHMC(SELF):

RETURN SELF.HMC

DEF setHMC(SELF,HMC):

SELF.HMC = HMC

DEF checkRepition(SELF):

RETURN any(value>=3 FOR value in SELF.history.values())

DEF addtoHistory(SELF,position):

KEY = SELF.c.pos2KEY(position)

SELF.history[KEY] = SELF.history.get(KEY,0) + 1

DEF gethistory(SELF):

RETURN SELF.history

DEF clone(SELF):

SELF.PLAYER,

copy.deepcopy(SELF.castling), # Independent copy

SELF.EnP,

SELF.HMC)

RETURN clone

CLASS Shades:

DEF \_\_init\_\_(SELF,image,coord):

SELF.image = image

SELF.pos = coord

DEF getInfo(SELF):

RETURN [SELF.image,SELF.pos]

CLASS Piece:

DEF \_\_init\_\_(SELF,pieceinfo,chess\_coord,square\_width, square\_height):

PIECE = pieceinfo[0]

COLOR = pieceinfo[1]

IF piece=='K':

INDEX = 0

ELIF piece=='Q':

INDEX = 1

ELIF piece=='B':

INDEX = 2

ELIF piece == 'N':

INDEX = 3

ELIF piece == 'R':

INDEX = 4

ELIF piece == 'P':

INDEX = 5

left\_x =square\_width\*INDEX

IF COLOR == 'w':

left\_y = 0

ELSE:

left\_y = square\_height

SELF.pieceinfo = pieceinfo

SELF.subsection = (left\_x,left\_y,square\_width,square\_height)

SELF.chess\_coord = chess\_coord

SELF.pos = (-1,-1)

DEF getInfo(SELF):

RETURN [SELF.chess\_coord, SELF.subsection,SELF.pos]

DEF setpos(SELF,pos):

SELF.pos = pos

DEF getpos(SELF):

RETURN SELF.pos

DEF setcoord(SELF,coord):

SELF.chess\_coord = coord

DEF isOccupied(SELF,BOARD,x,y):

IF BOARD[int(y)][int(x)] == 0:

RETURN False

RETURN True

DEF isOccupiedby(SELF,BOARD,x,y,COLOR):

IF BOARD[y][x] == 0:

RETURN False

IF BOARD[y][x][1] == COLOR[0]:

RETURN True

RETURN False

DEF filterbyCOLOR(SELF,BOARD,LISTofTuples,COLOR):

filtered\_LIST = []

FOR pos in LISTofTuples: x = pos[0] y = pos[1]

IF x>=0 and x<=7 and y>=0 and y<=7 and NOT SELF.isOccupiedby(BOARD,x,y,COLOR):

FILTERED\_LIST.append(pos)

RETURN filtered\_LIST

DEF lookFOR(SELF,BOARD,piece):

FOR row in range(8):

FOR col in range(8):

IF BOARD[row][col] == piece: x = col, y = row

LISTOFLOCATIONS.append((x,y))

RETURN LISTofLocations

DEF isAttackedby(SELF,position,target\_x,target\_y,COLOR):

BOARD = position.getBOARD()

COLOR = COLOR[0]

LISTOFATTACKEDSQUARES = []

FOR x in range(8):

FOR y in range(8):

IF BOARD[y][x]!=0 and BOARD[y][x][1]==COLOR:

LISTOFATTACKEDSQUARES.extend(

SELF.findPossibleSquares(position,x,y,True)) #The true argument

RETURN (target\_x,target\_y) in LISTofAttackedSquares

DEF findPossibleSquares(SELF,position,x,y,AttackSearch=False):

BOARD = position.getBOARD()

PLAYER = position.getPLAYER()

CASTLING\_RIGHTS = position.getCastleRights()

ENP\_Target = position.getEnP()

PIECE = BOARD[y][x][0] #Pawn, rook, etc.

COLOR = BOARD[y][x][1] #w or b.

ENEMY\_COLOR = SELF.opp(COLOR)

LISTOFTUPLES = [] #Holds LIST of attacked squares.

IF piece == 'P': #The piece is a pawn.

IF COLOR=='w': #The piece is white

IF NOT SELF.isOccupied(BOARD,x,y-1) and NOT AttackSearch:

LISTOFTUPLES.append((x,y-1))

IF y == 6 and NOT SELF.isOccupied(BOARD,x,y-2):

LISTOFTUPLES.append((x,y-2))

IF x!=0 and SELF.isOccupiedby(BOARD,x-1,y-1,'black'):

LISTOFTUPLES.append((x-1,y-1))

IF x!=7 and SELF.isOccupiedby(BOARD,x+1,y-1,'black'):

LISTOFTUPLES.append((x+1,y-1))

IF EnP\_Target!=-1: #There is a possible en pasant target:

IF EnP\_Target == (x-1,y-1) or EnP\_Target == (x+1,y-1):

LISTOFTUPLES.append(EnP\_Target)

ELIF COLOR=='b': #The piece is black, same as above but opposite side.

IF NOT SELF.isOccupied(BOARD,x,y+1) and NOT AttackSearch:

LISTOFTUPLES.append((x,y+1))

IF y == 1 and NOT SELF.isOccupied(BOARD,x,y+2):

LISTOFTUPLES.append((x,y+2))

IF x!=0 and SELF.isOccupiedby(BOARD,x-1,y+1,'white'):

LISTOFTUPLES.append((x-1,y+1))

IF x!=7 and SELF.isOccupiedby(BOARD,x+1,y+1,'white'):

LISTOFTUPLES.append((x+1,y+1))

IF EnP\_Target == (x-1,y+1) or EnP\_Target == (x+1,y+1):

LISTOFTUPLES.append(EnP\_Target)

ELIF piece == 'R': #The piece is a rook.

FOR i in [-1,1]: kx = x #This variable stores the x coordinate being looked at.

WHILE True: kx = kx + i #Searching left or right

IF kx<=7 and kx>=0: #Making sure we're still in BOARD.

IF NOT SELF.isOccupied(BOARD,kx,y):

LISTOFTUPLES.append((kx,y))

ELSE:

IF SELF.isOccupiedby(BOARD,kx,y,enemy\_COLOR):

LISTofTuples.append((kx,y))

BREAK

ELSE: #We have exceeded the limits of the BOARD

BREAK

FOR i in [-1,1]:

ky = y

WHILE True:

ky = ky + i

IF ky<=7 and ky>=0:

IF NOT SELF.isOccupied(BOARD,x,ky):

LISTOFTUPLES.append((x,ky))

ELSE:

IF SELF.isOccupiedby(BOARD,x,ky,enemy\_COLOR):

LISTOFTUPLES.append((x,ky))

BREAK

ELSE:

BREAK

ELIF piece == 'N': #The piece is a knight.

FOR dx in [-2,-1,1,2]:

IF abs(dx)==1:

sy = 2

ELSE:

sy = 1

FOR dy in [-sy,+sy]:

LISTOFTUPLES.append((x+dx,y+dy))

LISTOFTUPLES = SELF.filterbyCOLOR(BOARD,LISTofTuples,COLOR)

ELIF piece == 'B': # A bishop.

FOR dx in [-1,1]: #Allow two directions in x.

FOR dy in [-1,1]: #Similarly, up and down FOR y.

kx = x #These varibales store the coordinates of the square being

ky = y

WHILE True: #loop till broken.

kx = kx + dx #change x

ky = ky + dy #change y

IF kx<=7 and kx>=0 and ky<=7 and ky>=0:

IF NOT SELF.isOccupied(BOARD,kx,ky):

LISTOFTUPLES.append((kx,ky))

ELSE:

IF SELF.isOccupiedby(BOARD,kx,ky,enemy\_COLOR):

LISTofTuples.append((kx,ky))

BREAK

ELSE:

BREAK

ELIF piece == 'Q': #A queen

BOARD[y][x] = 'R' + COLOR

LIST\_rook = SELF.findPossibleSquares(position,x,y,True)

BOARD[y][x] = 'B' + COLOR

LIST\_bishop = SELF.findPossibleSquares(position,x,y,True)

LISTOFTUPLES = LIST\_rook + LIST\_bishop

BOARD[y][x] = 'Q' + COLOR

ELIF piece == 'K': # A king!

FOR dx in [-1,0,1]:

FOR dy in [-1,0,1]:

LISTOFTUPLES.append((x+dx,y+dy))

LISTOFTUPLES = SELF.filterbyCOLOR(BOARD,LISTofTuples,COLOR)

IF NOT AttackSearch:

RIGHT = castling\_rights[PLAYER]

IF (right[0] and #has right to castle

BOARD[y][7]!=0 and #The rook square is NOT empty

BOARD[y][7][0]=='R' and #There is a rook at the appropriate place

NOT SELF.isOccupied(BOARD,x+1,y) and #The square on its right is empty

NOT SELF.isOccupied(BOARD,x+2,y) and #The second square beyond is also empty

NOT SELF.isAttackedby(position,x,y,enemy\_COLOR) and #The king isn't under atack

NOT SELF.isAttackedby(position,x+1,y,enemy\_COLOR) and #Or the path through which

NOT SELF.isAttackedby(position,x+2,y,enemy\_COLOR)):#it will move

LISTofTuples.append((x+2,y))

IF (right[1] and #has right to castle

BOARD[y][0]!=0 and #The rook square is NOT empty

BOARD[y][0][0]=='R' and #The rook square is NOT empty

NOT SELF.isOccupied(BOARD,x-1,y)and #The square on its left is empty

NOT SELF.isOccupied(BOARD,x-2,y)and #The second square beyond is also empty

NOT SELF.isOccupied(BOARD,x-3,y) and #And the one beyond.

NOT SELF.isAttackedby(position,x,y,enemy\_COLOR) and #The king isn't under atack

NOT SELF.isAttackedby(position,x-1,y,enemy\_COLOR) and #Or the path through which

NOT SELF.isAttackedby(position,x-2,y,enemy\_COLOR)):#it will move

LISTOFTUPLES.append((x-2,y)) #Let castling be an option.

IF NOT AttackSearch:

new\_LIST = []

FOR tupleq in LISTofTuples:

x2 = int(tupleq[0])

y2 = int(tupleq[1])

TEMP\_pos = position.clone()

SELF.makemove(TEMP\_pos,x,y,x2,y2)

IF NOT SELF.isCheck(TEMP\_pos,COLOR):

new\_LIST.append(tupleq)

LISTofTuples = new\_LIST

RETURN LISTofTuples

x = int(x)

y = int(y)

x2 = int(x2)

y2 = int(y2)

BOARD = position.getBOARD()

PIECE = BOARD[y][x]

IF PIECE ==0:

RETURN

PIECE =piece[0]

COLOR = BOARD[y][x][1]

PLAYER = position.getPLAYER()

CASTLING\_RIGHTS = position.getCastleRights()

ENP\_Target = position.getEnP()

HALF\_MOVE\_CLOCK = position.getHMC()

IF SELF.isOccupied(BOARD,x2,y2) or piece=='P':

HALF\_MOVE\_CLOCK = 0

ELSE:

HALF\_MOVE\_CLOCK += 1

BOARD[y2][x2] = BOARD[y][x]

BOARD[y][x] = 0

IF piece == 'K':

castling\_rights[PLAYER] = [False,False]

IF abs(x2-x) == 2:

IF COLOR=='w':

l = 7

ELSE:

l = 0

IF x2>x:

BOARD[l][5] = 'R'+COLOR

BOARD[l][7] = 0

ELSE:

BOARD[l][3] = 'R'+COLOR

BOARD[l][0] = 0

IF piece=='R':

IF x==0 and y==0:

CASTLING\_RIGHTS[1][1] = False

ELIF x==7 and y==0:

CASTLING\_RIGHTS[1][0] = False

ELIF x==0 and y==7:

CASTLING\_RIGHTS[0][1] = False

ELIF x==7 and y==7:

CASTLING\_RIGHTS[0][0] = False

IF piece == 'P':

IF EnP\_Target == (x2,y2):

IF COLOR=='w':

BOARD[y2+1][x2] = 0

ELSE:

BOARD[y2-1][x2] = 0

IF abs(y2-y)==2:

EnP\_Target = (x,(y+y2)/2)

ELSE:

EnP\_Target = -1

IF y2==0:

BOARD[y2][x2] = 'Qw'

ELIF y2 == 7:

BOARD[y2][x2] = 'Qb'

ELSE:

ENP\_Target = -1

PLAYER = 1 – PLAYER

POSITION.setPLAYER(PLAYER)

POSITION.setCastleRights(castling\_rights)

POSITION.setEnP(EnP\_Target)

POSITION.setHMC(half\_move\_clock)

DEF opp(SELF,COLOR):

COLOR = COLOR[0]

IF COLOR == 'w':

OPPCOLOR = 'b'

ELSE:

OPPCOLOR = 'w'

RETURN oppCOLOR

DEF isCheck(SELF,position,COLOR):

BOARD = position.getBOARD()

COLOR = COLOR[0]

ENEMY = SELF.opp(COLOR)

PIECE = 'K' + COLOR

x,y = SELF.lookFOR(BOARD,piece)[0]

RETURN SELF.isAttackedby(position,x,y,enemy)

DEF isCheckmate(SELF,position,COLOR=-1):

IF COLOR==-1:

RETURN SELF.isCheckmate(position,'white') or SELF.isCheckmate(position,'b')

COLOR = COLOR[0]

IF SELF.isCheck(position,COLOR) and SELF.allMoves(position,COLOR)==[]:

RETURN True

RETURN False

DEF isStalemate(SELF,position):

PLAYER = position.getPLAYER()

IF PLAYER==0:

COLOR = 'w'

ELSE:

COLOR = 'b'

RETURN True

RETURN False

DEF getallpieces(SELF,position,COLOR):

BOARD = position.getBOARD()

LISTofpos = []

FOR j in range(8):

FOR i in range(8):

IF SELF.isOccupiedby(BOARD,i,j,COLOR):

LISTofpos.append((i,j))

RETURN LISTofpos

DEF allMoves(SELF,position, COLOR):

IF COLOR==1:

COLOR = 'white'

ELIF COLOR ==-1:

COLOR = 'black'

COLOR = COLOR[0]

LISTofpieces = SELF.getallpieces(position,COLOR)

FOR pos in LISTofpieces:

targets = SELF.findPossibleSquares(position,pos[0],pos[1])

FOR target in targets:

MOVES.append([pos,target])

RETURN moves

DEF pos2KEY(SELF,position):

BOARD = position.getBOARD()

BOARDTuple = []

FOR row in BOARD:

BOARDTuple.append(tuple(row))

BOARDTuple = tuple(BOARDTuple)

RIGHTS = position.getCastleRights()

TUPLERIGHTS = (tuple(rights[0]),tuple(rights[1]))

KEY = (BOARDTuple,position.getPLAYER(),tuplerights)

#RETURN the KEY:

RETURN KEY

###########################////////AI RELATED FUNCTIONS\\\\\\\\\\############################

CLASS AI:

DEF \_\_init\_\_(SELF):

SELF.c=Commands()

DEF negamax( SELF,position,depth,alpha,beta,COLORsign,bestMoveRETURN,openings,searched,root=True):

IF root:

KEY = SELF.c.pos2KEY(position)

IF KEY in openings:

bestMoveRETURN[:] = random.choice(openings[KEY])

RETURN

IF depth==0:

RETURN COLORsign\*SELF.evaluate(position)

moves = SELF.c.allMoves(position, COLORsign)

IF moves==[]:

RETURN COLORsign\*SELF.evaluate(position)

IF root:

bestMove = moves[0]

bestValue = -100000

FOR move in moves:

newpos = position.clone()

SELF.c.makemove(newpos,move[0][0],move[0][1],move[1][0],move[1][1])

KEY = SELF.c.pos2KEY(newpos)

IF KEY in searched:

value = searched[KEY]

ELSE:

value = -SELF.negamax(newpos,depth-1, -beta,-alpha,-COLORsign,[],openings,searched,False)

searched[KEY] = value

IF value>bestValue:

bestValue = value

IF root:

bestMove = move

alpha = max(alpha,value)

IF alpha>=beta:

BREAK

IF root:

searched = {}

bestMoveRETURN[:] = bestMove

RETURN

RETURN bestValue

DEF evaluate(SELF,position):

IF SELF.c.isCheckmate(position,'white'):

RETURN -20000

IF SELF.c.isCheckmate(position,'black'):

RETURN 20000

BOARD = position.getBOARD()

flatBOARD = [x FOR row in BOARD FOR x in row]

c = Counter(flatBOARD)

Qw = c['Qw']

Qb = c['Qb']

Rw = c['Rw']

Rb = c['Rb']

Bw = c['Bw']

Bb = c['Bb']

Nw = c['Nw']

Nb = c['Nb']

Pw = c['Pw']

Pb = c['Pb']

whiteMaterial = 9\*Qw + 5\*Rw + 3\*Nw + 3\*Bw + 1\*Pw

blackMaterial = 9\*Qb + 5\*Rb + 3\*Nb + 3\*Bb + 1\*Pb

numofmoves = len(position.gethistory())

gamephase = 'opening'

IF numofmoves>40 or (whiteMaterial<14 and blackMaterial<14):

gamephase = 'ending'

Dw = SELF.doubledPawns(BOARD,'white')

Db = SELF.doubledPawns(BOARD,'black')

Sw = SELF.blockedPawns(BOARD,'white')

Sb = SELF.blockedPawns(BOARD,'black')

Iw = SELF.isolatedPawns(BOARD,'white')

Ib = SELF.isolatedPawns(BOARD,'black')

evaluation1 = 900\*(Qw - Qb) + 500\*(Rw - Rb) +330\*(Bw-Bb

)+320\*(Nw - Nb) +100\*(Pw - Pb) +-30\*(Dw-Db + Sw-Sb + Iw- Ib

)

evaluation2 = SELF.pieceSquareTable(flatBOARD,gamephase)

evaluation = evaluation1 + evaluation2

RETURN evaluation

DEF pieceSquareTable(SELF,flatBOARD,gamephase):

p=PieceTable()

score = 0

FOR i in range(64):

IF flatBOARD[i]==0:

CONTINUE

piece = flatBOARD[i][0]

COLOR = flatBOARD[i][1]

sign = +1:

IF COLOR=='b':

i = ((7-i)//8)\*8 + i%8

sign = -1

IF piece=='P':

score += sign\*p.pawn\_table[i]

ELIF piece=='N':

score+= sign\*p.knight\_table[i]

ELIF piece=='B':

score+=sign\*p.bishop\_table[i]

ELIF piece=='R':

score+=sign\*p.rook\_table[i]

ELIF piece=='Q':

score+=sign\*p.queen\_table[i]

ELIF piece=='K':

IF gamephase=='opening':

score+=sign\*p.king\_table[i]

ELSE:

score+=sign\*p.king\_endgame\_table[i]

RETURN score

DEF doubledPawns(SELF,BOARD,COLOR):

COLOR = COLOR[0]

LISTofpawns = SELF.c.lookFOR(BOARD,'P'+COLOR)

repeats = 0

TEMP = []

FOR pawnpos in LISTofpawns:

IF pawnpos[0] in TEMP:

repeats = repeats + 1

ELSE:

TEMP.append(pawnpos[0])

RETURN repeats

DEF blockedPawns(SELF,BOARD,COLOR):

COLOR = COLOR[0]

LISTofpawns = SELF.c.lookFOR(BOARD,'P'+COLOR)

blocked = 0

FOR pawnpos in LISTofpawns:

IF ((COLOR=='w' and SELF.c.isOccupiedby(BOARD,pawnpos[0],pawnpos[1]-1,

'black'))

or (COLOR=='b' and SELF.c.isOccupiedby(BOARD,pawnpos[0],pawnpos[1]+1,

'white'))):

blocked = blocked + 1

RETURN blocked

DEF isolatedPawns(SELF,BOARD,COLOR):

COLOR = COLOR[0]

LISTofpawns = SELF.c.lookFOR(BOARD,'P'+COLOR)

xLIST = [x FOR (x,y) in LISTofpawns]

isolated = 0

FOR x in xLIST:

IF x!=0 and x!=7:

IF x-1 NOT in xLIST and x+1 NOT in xLIST:

isolated+=1

ELIF x==0 and 1 NOT in xLIST:

isolated+=1

ELIF x==7 and 6 NOT in xLIST:

isolated+=1

RETURN isolated

CLASS BOARD:

DEF \_\_init\_\_(SELF):

SELF.create\_BOARD()

DEF create\_BOARD(SELF):

SELF.chess=[[0]\*8 FOR i in range(8)]

LIST\_w=['Rw','Nw','Bw','Qw','Kw','Bw','Nw','Rw']

LIST\_b=['Rb','Nb','Bb','Qb','Kb','Bb','Nb','Rb']

FOR i in range(2):

FOR j in range (8):

IF i==0:

SELF.chess[i][j]=LIST\_b[j]

ELSE:

SELF.chess[i][j]='Pb'

FOR i in range(6,8):

FOR j in range (8):

IF i==7:

SELF.chess[i][j]=LIST\_w[j]

ELSE:

SELF.chess[i][j]='Pw'

DEF getChess(SELF):

RETURN SELF.chess

CLASS PieceTable:

DEF \_\_init\_\_(SELF):

SELF.pawn\_table = [ 0, 0, 0, 0, 0, 0, 0, 0,

50, 50, 50, 50, 50, 50, 50, 50,

10, 10, 20, 30, 30, 20, 10, 10,

5, 5, 10, 25, 25, 10, 5, 5,

0, 0, 0, 20, 20, 0, 0, 0,

5, -5,-10, 0, 0,-10, -5, 5,

5, 10, 10,-20,-20, 10, 10, 5,

0, 0, 0, 0, 0, 0, 0, 0]

SELF.knight\_table = [-50,-40,-30,-30,-30,-30,-40,-50,

-40,-20, 0, 0, 0, 0,-20,-40,

-30, 0, 10, 15, 15, 10, 0,-30,

-30, 5, 15, 20, 20, 15, 5,-30,

-30, 0, 15, 20, 20, 15, 0,-30,

-30, 5, 10, 15, 15, 10, 5,-30,

-40,-20, 0, 5, 5, 0,-20,-40,

-50,-90,-30,-30,-30,-30,-90,-50]

SELF.bishop\_table = [-20,-10,-10,-10,-10,-10,-10,-20,

-10, 0, 0, 0, 0, 0, 0,-10,

-10, 0, 5, 10, 10, 5, 0,-10,

-10, 5, 5, 10, 10, 5, 5,-10,

-10, 0, 10, 10, 10, 10, 0,-10,

-10, 10, 10, 10, 10, 10, 10,-10,

-10, 5, 0, 0, 0, 0, 5,-10,

-20,-10,-90,-10,-10,-90,-10,-20]

SELF.rook\_table = [0, 0, 0, 0, 0, 0, 0, 0,

5, 10, 10, 10, 10, 10, 10, 5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

0, 0, 0, 5, 5, 0, 0, 0]

SELF.queen\_table = [-20,-10,-10, -5, -5,-10,-10,-20,

-10, 0, 0, 0, 0, 0, 0,-10,

-10, 0, 5, 5, 5, 5, 0,-10,

-5, 0, 5, 5, 5, 5, 0, -5,

0, 0, 5, 5, 5, 5, 0, -5,

-10, 5, 5, 5, 5, 5, 0,-10,

-10, 0, 5, 0, 0, 0, 0,-10,

-20,-10,-10, 70, -5,-10,-10,-20]

SELF.king\_table = [-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-20,-30,-30,-40,-40,-30,-30,-20,

-10,-20,-20,-20,-20,-20,-20,-10,

20, 20, 0, 0, 0, 0, 20, 20,

20, 30, 10, 0, 0, 10, 30, 20]

SELF.king\_endgame\_table = [-50,-40,-30,-20,-20,-30,-40,-50,

-30,-20,-10, 0, 0,-10,-20,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-30, 0, 0, 0, 0,-30,-30,

-50,-30,-30,-30,-30,-30,-30,-50]

##############################////////GUI FUNCTIONS\\\\\\\\\\\\\#############################

#########MAIN FUNCTION####################################################

CLASS GUI:

DEF \_\_init\_\_(SELF):

SELF.BOARD = BOARD().getChess()

SELF.c = Commands()

SELF.a = AI()

SELF.PLAYER = 0 #This is the PLAYER that makes the next move. 0 is white, 1 is black

SELF.castling\_rights = [[True, True],[True, True]]

SELF.En\_Passant\_Target = -1 #This variable will store a coordinate IF there is a square that can be

SELF.half\_move\_clock = 0 #This variable stores the number of reversible moves that have been played so far.

SELF.position = GamePosition(SELF.BOARD,SELF.PLAYER,SELF.castling\_rights,SELF.En\_Passant\_Target

,SELF.half\_move\_clock)

PYGAME.init()

SELF.size = (640, 640)

SELF.screen = PYGAME.display.set\_mode(SELF.size)

PYGAME.display.set\_caption("Chess Game")

SELF.game\_icon = PYGAME.image.load('newMedia/ChessImage.png')

PYGAME.display.set\_icon(SELF.game\_icon)

SELF.media()

SELF.bg = (49, 60, 43)

SELF.startPage = PYGAME.Surface(SELF.size)

SELF.startPage.fill(SELF.bg)

SELF.dIFfPage = PYGAME.Surface(SELF.size)

SELF.dIFfPage.fill(SELF.bg)

SELF.flipPage = PYGAME.Surface(SELF.size)

SELF.flipPage.fill(SELF.bg)

SELF.SELECTPage = PYGAME.Surface(SELF.size)

SELF.SELECTPage.fill(SELF.bg)

SELF.COLORPage = PYGAME.Surface(SELF.size)

SELF.COLORPage.fill(SELF.bg)

# Stored [ x , y , width , height ] of buttons

SELF.buttons = {

1: [460-275, 380-15, 280, 75],

2: [460-275, 470-15, 280, 75],

3: [325-275, 280-15, 250, 250],

4: [625-275, 280-15, 250, 250],

5: [309-275, 250-15, 180, 180],

6: [509-275, 250-15, 180, 180],

7: [709-275, 250-15, 180, 180]

}

SELF.dIFfMenu = -1

SELF.SELECT = -1

SELF.levEL = None

SELF.TEMP = None

SELF.box = PYGAME.image.load('newMedia/box.png')

SELF.box = PYGAME.transFORm.scale(SELF.box, (640, 640))

SELF.screen.blit(SELF.box,(0,0))

PYGAME.mixer.Sound.play(SELF.wELcome\_sound)

clock = PYGAME.time.Clock() # HELps controlling fps of the game.

SELF.initialize()

PYGAME.display.update()

#########################INFINITE LOOP#####################################

WHILE NOT SELF.gameEnded:

IF SELF.isMenu:

IF SELF.isAI==-1:

SELF.startMenu()

ELIF SELF.isAI==True:

IF SELF.dIFfMenu == -1:

SELF.play1Menu\_A()

ELIF SELF.dIFfMenu == 1:

SELF.play1Menu\_B()

IF SELF.SELECT == 1 and SELF.TEMP == None:

SELF.SELECTMenu()

ELIF SELF.isAI==False:

SELF.play2Menu()

IF SELF.isFlip!=-1 and SELF.SELECT == 2 :

SELF.call\_BOARD()

CONTINUE

ELIF SELF.isFlip!=-1 and SELF.SELECT == 3 :

SELF.call\_BOARD()

PYGAME.mixer.Sound.play(SELF.instructions\_sound)

CONTINUE

IF SELF.isFlip!=-1 and SELF.TEMP == -1 :

SELF.call\_BOARD()

CONTINUE

FOR event in PYGAME.event.get():

IF event.type == QUIT:

SELF.gameEnded = True

PYGAME.mixer.Sound.play(SELF.exit\_sound)

BREAK

IF event.type == MOUSEBUTTONUP:

SELF.onClick()

PYGAME.display.update()

clock.tick(10)

CONTINUE

SELF.numm+=1

IF SELF.isAIThink and SELF.numm%10==0:

SELF.Thinking()

FOR event in PYGAME.event.get():

#Deal with all the user inputs:

IF event.type==QUIT:

SELF.gameEnded = True

PYGAME.mixer.Sound.play(SELF.exit\_sound)

BREAK

IF SELF.chessEnded or SELF.isTransition or SELF.isAIThink:

CONTINUE

IF SELF.SELECT<=2:

IF NOT SELF.isDown and event.type == MOUSEBUTTONDOWN:

pos = PYGAME.mouse.get\_pos()

IF pos[0] in range(0,640) and pos[1] in range(0,640):

chess\_coord = SELF.PIXEL\_coord\_to\_chess(pos)

x = chess\_coord[0]

y = chess\_coord[1]

IF NOT SELF.c.isOccupiedby(SELF.BOARD,x,y,'wb'[SELF.PLAYER]):

CONTINUE

dragPiece = SELF.getPiece(chess\_coord)

LISTofTuples = SELF.c.findPossibleSquares(SELF.position,x,y)

SELF.createShades(LISTofTuples)

IF dragPiece:

IF ((dragPiece.pieceinfo[0]=='K') and

(SELF.c.isCheck(SELF.position,'white') or SELF.c.isCheck(SELF.position,'black'))):

None

ELSE:

SELF.LISTofShades.append(Shades(SELF.greenbox\_image,(x,y)))

SELF.isDown = True

IF (SELF.isDown or SELF.isClicked) and event.type == MOUSEBUTTONUP:

SELF.isDown = False

IF dragPiece:

dragPiece.setpos((-1,-1))

pos = PYGAME.mouse.get\_pos()

chess\_coord = SELF.PIXEL\_coord\_to\_chess(pos)

x2 = chess\_coord[0]

y2 = chess\_coord[1]

SELF.isTransition = False

IF (x,y)==(x2,y2): #NO dragging occured

IF NOT SELF.isClicked: #NOThing had been clicked previously

SELF.isClicked = True

SELF.prevPos = (x,y) #Store it so next time we know the origin

ELSE: #Something had been clicked previously

x,y = SELF.prevPos

IF (x,y)==(x2,y2): #User clicked on the same square again.

SELF.isClicked = False

SELF.createShades([])

ELSE:

IF SELF.c.isOccupiedby(SELF.BOARD,x2,y2,'wb'[SELF.PLAYER]):

SELF.isClicked = True

SELF.prevPos = (x2,y2) #Store it

ELSE:

SELF.isClicked = False

window = Tk()

window.wm\_withdraw()

window.geometry("1x1+200+200")

tkinter.messagebox.showinfo(title="Invalid move",message="Invalid move",parent=window)

ELSE:

IF self.c.isOccupiedby(self.board,x2,y2,'wb'[self.player]):

self.isClicked = True

self.prevPos = (x2,y2) #Store it

ELSE:

self.isClicked = False

IF NOT (x2,y2) in listofTuples:

window = Tk()

window.wm\_withdraw()

window.geometry("1x1+200+200")

tkinter.messagebox.showinfo(title="Invalid move",message="Invalid move",parent=window)

SELF.createShades([])

SELF.isTransition = True #Possibly IF the move was valid.

IF NOT (x2,y2) in LISTofTuples:

SELF.isTransition = False

CONTINUE

IF SELF.isRecord:

KEY = SELF.c.pos2KEY(SELF.position)

IF [(x,y),(x2,y2)] NOT in SELF.openings[KEY]:

SELF.openings[KEY].append([(x,y),(x2,y2)])

SELF.c.makemove(SELF.position,x,y,x2,y2)

SELF.prevMove = [x,y,x2,y2]

SELF.PLAYER = SELF.position.getPLAYER()

IF SELF.PLAYER == 1:

PYGAME.mixer.Sound.play(SELF.piece\_sound)

ELSE:

PYGAME.mixer.Sound.play(SELF.piece\_sound)

SELF.position.addtoHistory(SELF.position)

HMC = SELF.position.getHMC()

IF HMC>=100 or SELF.c.isStalemate(SELF.position) or SELF.position.checkRepition():

SELF.isDraw = True

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position,'white'):

SELF.winner = 'b'

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position,'black'):

SELF.winner = 'w'

SELF.chessEnded = True

IF SELF.isAI and NOT SELF.chessEnded:

IF SELF.PLAYER==0:

COLORsign = 1

ELSE:

COLORsign = -1

SELF.bestMoveRETURN = []

SELF.move\_thread = threading.Thread(target = SELF.a.negamax,

args = (SELF.position,SELF.levEL,-1000000,1000000,COLORsign,SELF.bestMoveRETURN,SELF.openings,SELF.searched))

SELF.move\_thread.start()

SELF.isAIThink = True

dragPiece.setcoord((x2,y2))

IF NOT SELF.isTransition:

SELF.LISTofWhitePieces,SELF.LISTofBlackPieces = SELF.createPieces(SELF.BOARD)

ELSE:

movingPiece = dragPiece

origin = SELF.chess\_coord\_to\_PIXELs((x,y))

destiny = SELF.chess\_coord\_to\_PIXELs((x2,y2))

movingPiece.setpos(origin)

step = (destiny[0]-origin[0],destiny[1]-origin[1])

SELF.createShades([])

ELSE:

IF event.type == PYGAME.MOUSEBUTTONDOWN and event.button==1:

IF SELF.PLAYER==1:

SELF.letters\_dict = {'a': 7, 'b': 6, 'c': 5, 'd': 4, 'e': 3, 'f': 2, 'g': 1, 'h': 0}

SELF.numbers\_dict = {'1': 0, '2': 1, '3': 2, '4': 3, '5': 4, '6': 5, '7': 6, '8': 7}

with sr.Microphone() as source:

SELF.r.adjust\_FOR\_ambient\_noise(source)

PYGAME.mixer.Sound.play(SELF.SELECTpiece\_sound)

time.sleep(1.5)

TRY:

AUDIO = SELF.r.LISTen(source,timeout=2,phrase\_time\_limit=2)

PRINT("Recognizing...")

query = SELF.r.recognize\_google(AUDIO)

PRINT(f"User said: {query}\n")

VOICE = query.lower()

IF VOICE=='avon':

VOICE='a1'

ELIF VOICE == 'heetu' or VOICE=='hetu' or VOICE=='do' or VOICE =='tattoo' or VOICE =='airport' or VOICE =='tetu' or VOICE =='edu':

VOICE='a2'

ELIF VOICE == 'a tree' or VOICE=='83':

VOICE='a3'

ELIF VOICE == 'krrish 4':

VOICE='a4'

ELIF VOICE=='beFORe':

VOICE='b4'

ELIF VOICE=='bittu' or VOICE=='titu':

VOICE='b2'

ELIF VOICE=='ba' or VOICE=='b.ed':

VOICE='b8'

ELIF VOICE=='shivan' or VOICE=='shiva' or VOICE=='civil':

VOICE='c1'

ELIF VOICE=='ceat':

VOICE='c8'

ELIF VOICE=='deewan' or VOICE=='d 1' or VOICE=='devon' or VOICE=='devil':

VOICE='d1'

ELIF VOICE=='even' or VOICE=='evil' or VOICE=='evan' or VOICE=='yuvan' or VOICE=='t1':

VOICE='e1'

ELIF VOICE=='youtube' or VOICE=='tu':

VOICE='e2'

ELIF VOICE=='mi 4':

VOICE='e4'

ELIF VOICE=='mi 5':

VOICE='e5'

ELIF VOICE=='8':

VOICE='e8'

ELIF VOICE=='jivan':

VOICE='g1'

ELIF VOICE=='jeetu' or VOICE=='jitu':

VOICE='g2'

ELIF VOICE=='zefo':

VOICE='g4'

ELIF VOICE=='quit' or VOICE =='end' or VOICE == 'close' or VOICE=='stop' or VOICE=='friend' or VOICE=='top' or VOICE=='finish' or VOICE=='and':

PYGAME.mixer.Sound.play(SELF.exit\_sound)

SELF.gameEnded=True

IF len(VOICE) == 2:

letter = VOICE[0]

number = VOICE[1]

IF letter=='v':

letter='b'

ELIF letter=='s':

letter='h'

IF letter in SELF.letters\_dict.KEYs() and number in SELF.numbers\_dict.KEYs():

PRINT(SELF.letters\_dict[letter], SELF.numbers\_dict[number])

chess\_coord = (SELF.letters\_dict[letter], SELF.numbers\_dict[number])

x = chess\_coord[0]

y = chess\_coord[1]

IF NOT SELF.c.isOccupiedby(SELF.BOARD, x, y, 'wb'[SELF.PLAYER]):

CONTINUE

dragPiece = SELF.getPiece(chess\_coord)

LISTofTuples = SELF.c.findPossibleSquares(SELF.position, x, y)

SELF.createShades(LISTofTuples)

IF dragPiece:

IF ((dragPiece.pieceinfo[0] == 'K') and

(SELF.c.isCheck(SELF.position, 'white') or SELF.c.isCheck(SELF.position,

'black'))):

None

ELSE:

SELF.LISTofShades.append(Shades(SELF.greenbox\_image, (x, y)))

SELF.piece\_SELECTed\_by\_VOICE = True

EXCEPT sr.UnknownValueError:

PYGAME.mixer.Sound.play(SELF.repeat\_sound)

EXCEPT sr.RequestError:

PYGAME.mixer.Sound.play(SELF.requesterror\_sound)

EXCEPT EXCEPTion:

PYGAME.mixer.Sound.play(SELF.repeat\_sound)

ELIF SELF.piece\_SELECTed\_by\_VOICE and event.type==PYGAME.MOUSEBUTTONDOWN and event.button==3 :

SELF.piece\_SELECTed\_by\_VOICE = False

with sr.Microphone() as source:

WHILE True:

SELF.r.adjust\_FOR\_ambient\_noise(source)

PYGAME.mixer.Sound.play(SELF.destination\_sound)

time.sleep(1.5)

TRY:

AUDIO = SELF.r.LISTen(source,timeout=2,phrase\_time\_limit=2)

PRINT("Recognizing...")

query2 = SELF.r.recognize\_google(AUDIO)

PRINT(f"User said: {query2}\n")

VOICE2 = query2.lower()

IF VOICE2=='avon':

VOICE2='a1'

ELIF VOICE2 == 'heetu' or VOICE2=='hetu' or VOICE2=='do' or VOICE2 =='tattoo' or VOICE2 =='airport' or VOICE2 =='tetu' or VOICE2 =='edu':

VOICE2='a2'

ELIF VOICE2 == 'a tree' or VOICE2=='83':

VOICE2='a3'

ELIF VOICE2 == 'krrish 4':

VOICE2='a4'

ELIF VOICE2=='beFORe':

VOICE2='b4'

ELIF VOICE2=='bittu' or VOICE2=='titu':

VOICE2='b2'

ELIF VOICE2=='ba' or VOICE2=='b.ed':

VOICE2='b8'

ELIF VOICE2=='shivan' or VOICE2=='shiva' or VOICE2=='civil':

VOICE2='c1'

ELIF VOICE2=='ceat':

VOICE2='c8'

ELIF VOICE2=='deewan' or VOICE2=='d 1' or VOICE2=='devon' or VOICE2=='devil':

VOICE2='d1'

ELIF VOICE2=='even' or VOICE2=='evil' or VOICE2=='evan' or VOICE2=='yuvan' or VOICE2=='t1':

VOICE='e1'

ELIF VOICE2=='youtube' or VOICE2=='tu':

VOICE2='e2'

ELIF VOICE2=='mi 4':

VOICE2='e4'

ELIF VOICE2=='mi 5':

VOICE2='e5'

ELIF VOICE2=='8':

VOICE2='e8'

ELIF VOICE2=='jivan':

VOICE2='g1'

ELIF VOICE2=='jeetu' or VOICE2=='jitu':

VOICE2='g2'

ELIF VOICE2=='zefo':

VOICE2='g4'

ELIF VOICE2 == 'quit' or VOICE2 == 'end' or VOICE2 == 'close' or VOICE2=='stop' or VOICE2=='friend' or VOICE2=='top' or VOICE2=='finish' or VOICE2=='and':

PYGAME.mixer.Sound.play(SELF.exit\_sound)

SELF.gameEnded=True

BREAK

IF len(VOICE2) == 2:

letter = VOICE2[0]

number = VOICE2[1]

IF letter=='v':

letter='b'

ELIF letter=='s':

letter='h'

IF letter in SELF.letters\_dict.KEYs() and number in SELF.numbers\_dict.KEYs():

PRINT(SELF.letters\_dict[letter], SELF.numbers\_dict[number])

chess\_coord = (SELF.letters\_dict[letter], SELF.numbers\_dict[number])

x2 = chess\_coord[0]

y2 = chess\_coord[1]

SELF.isTransition = False

IF NOT (x2, y2) in LISTofTuples:

SELF.isTransition = False

Pygame.mixer.sound.play(self.invalid\_sound)

CONTINUE

IF SELF.isRecord:

KEY = SELF.c.pos2KEY(SELF.position)

IF [(x, y), (x2, y2)] NOT in SELF.openings[KEY]:

SELF.openings[KEY].append([(x, y), (x2, y2)])

SELF.c.makemove(SELF.position, x, y, x2, y2)

SELF.prevMove = [x, y, x2, y2]

SELF.PLAYER = SELF.position.getPLAYER()

IF SELF.PLAYER == 1:

PYGAME.mixer.Sound.play(SELF.piece\_sound)

ELSE:

PYGAME.mixer.Sound.play(SELF.piece\_sound)

SELF.position.addtoHistory(SELF.position)

HMC = SELF.position.getHMC()

IF HMC >= 100 or SELF.c.isStalemate(SELF.position) or SELF.position.checkRepition():

SELF.isDraw = True

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position, 'white'):

SELF.winner = 'b'

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position, 'black'):

SELF.winner = 'w'

SELF.chessEnded = True

IF SELF.isAI and NOT SELF.chessEnded:

IF SELF.PLAYER == 0:

COLORsign = 1

ELSE:

COLORsign = -1

SELF.bestMoveRETURN = []

SELF.move\_thread = threading.Thread(target=SELF.a.negamax,

args=(SELF.position, SELF.levEL, -1000000, 1000000,

COLORsign,

SELF.bestMoveRETURN, SELF.openings,

SELF.searched))

SELF.move\_thread.start()

SELF.isAIThink = True

dragPiece.setcoord((x2, y2))

IF NOT SELF.isTransition:

SELF.LISTofWhitePieces, SELF.LISTofBlackPieces = SELF.createPieces(SELF.BOARD)

ELSE:

movingPiece = dragPiece

origin = SELF.chess\_coord\_to\_PIXELs((x, y))

destiny = SELF.chess\_coord\_to\_PIXELs((x2, y2))

movingPiece.setpos(origin)

step = (destiny[0] - origin[0], destiny[1] - origin[1])

SELF.createShades([])

BREAK

EXCEPT sr.UnknownValueError:

PYGAME.mixer.Sound.play(SELF.repeat\_sound)

EXCEPT sr.RequestError:

PYGAME.mixer.Sound.play(SELF.requesterror\_sound)

EXCEPT EXCEPTion:

PYGAME.mixer.Sound.play(SELF.repeat\_sound)

#IF an animation is supposed to happen, make it happen:

IF SELF.isTransition:

p,q = movingPiece.getpos()

dx2,dy2 = destiny

n= 30.0

IF abs(p-dx2)<=abs(step[0]/n) and abs(q-dy2)<=abs(step[1]/n):

movingPiece.setpos((-1,-1))

SELF.LISTofWhitePieces,SELF.LISTofBlackPieces = SELF.createPieces(SELF.BOARD)

SELF.isTransition = False

SELF.createShades([])

ELSE:

movingPiece.setpos((p+step[0]/n,q+step[1]/n))

IF SELF.isDown:

m,k = PYGAME.mouse.get\_pos()

IF dragPiece:

dragPiece.setpos((m-SELF.square\_width/2,k-SELF.square\_height/2))

IF SELF.isAIThink and NOT SELF.isTransition:

IF NOT SELF.move\_thread.is\_alive():

SELF.isAIThink = False

SELF.createShades([])

IF len(SELF.bestMoveRETURN)==2:

[x,y],[x2,y2] = SELF.bestMoveRETURN

ELSE:

SELF.c.allMoves(SELF.position,COLOR)

SELF.c.makemove(SELF.position,x,y,x2,y2)

SELF.prevMove = [x,y,x2,y2]

SELF.PLAYER = SELF.position.getPLAYER()

HMC = SELF.position.getHMC()

SELF.position.addtoHistory(SELF.position)

IF HMC>=100 or SELF.c.isStalemate(SELF.position) or SELF.position.checkRepition():

SELF.isDraw = True

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position,'white'):

SELF.winner = 'b'

SELF.chessEnded = True

IF SELF.c.isCheckmate(SELF.position,'black'):

SELF.winner = 'w'

SELF.chessEnded = True

SELF.isTransition = True

movingPiece = SELF.getPiece((x,y))

ORIGIN = SELF.chess\_coord\_to\_PIXELs((x,y))

DESTINY = SELF.chess\_coord\_to\_PIXELs((x2,y2))

MOVINGPIECE.setpos(origin)

step = (destiny[0]-origin[0],destiny[1]-origin[1])

PYGAME.mixer.Sound.play(SELF.piece\_sound)

SELF.drawBOARD()

PYGAME.display.update()

clock.tick(60)

time.sleep(2)

PYGAME.quit()

IF SELF.isRecord:

file\_handle.seek(0)

pickle.dump(SELF.openings,file\_handle)

file\_handle.truncate()

file\_handle.close()

DEF DisplayPage(SELF, pageName):

SELF.SurfacesAtTop = SELF.SurfacesAtTop.fromKEYs(SELF.SurfacesAtTop, False)

SELF.screen.blit(SELF.Surfaces[pageName], (0, 0))

SELF.SurfacesAtTop[pageName] = True

DEF chess\_coord\_to\_PIXELs(SELF,chess\_coord):

x,y = chess\_coord

IF SELF.isAI:

IF SELF.AIPLAYER==0:

RETURN ((7-x)\*SELF.square\_width, (7-y)\*SELF.square\_height)

ELSE:

RETURN (x\*SELF.square\_width, y\*SELF.square\_height)

transition animation FOR

IF NOT SELF.isFlip or SELF.PLAYER==0 ^ SELF.isTransition:

RETURN (x\*SELF.square\_width, y\*SELF.square\_height)

ELSE:

RETURN ((7-x)\*SELF.square\_width, (7-y)\*SELF.square\_height)

DEF PIXEL\_coord\_to\_chess(SELF,PIXEL\_coord):

IF PIXEL\_coord[0] in range(0,640) and PIXEL\_coord[1] in range(0,640):

x,y = (PIXEL\_coord[0])//SELF.square\_width, (PIXEL\_coord[1])//SELF.square\_height

IF SELF.isAI:

IF SELF.AIPLAYER==0:

RETURN (7-x,7-y)

ELSE:

RETURN (x,y)

IF NOT SELF.isFlip or SELF.PLAYER==0 ^ SELF.isTransition:

RETURN (x,y)

ELSE:

RETURN (7-x,7-y)

DEF getPiece(SELF,chess\_coord):

FOR piece in SELF.LISTofWhitePieces+SELF.LISTofBlackPieces:

IF piece.getInfo()[0] == chess\_coord:

RETURN piece

DEF createPieces(SELF,BOARD):

SELF.LISTofWhitePieces = []

SELF.LISTofBlackPieces = []

FOR i in range(8):

FOR k in range(8):

IF BOARD[i][k]!=0:

p = Piece(BOARD[i][k],(k,i), SELF.square\_width, SELF.square\_height)

IF BOARD[i][k][1]=='w':

SELF.LISTofWhitePieces.append(p)

ELSE:

SELF.LISTofBlackPieces.append(p)

RETURN [SELF.LISTofWhitePieces,SELF.LISTofBlackPieces]

DEF createShades(SELF,LISTofTuples):

SELF.LISTofShades = []

IF SELF.isTransition:

RETURN

IF SELF.isDraw:

coord = SELF.c.lookFOR(SELF.BOARD,'Kw')[0]

shade = Shades(SELF.circle\_image\_yELlow,coord)

SELF.LISTofShades.append(shade)

coord = SELF.c.lookFOR(SELF.BOARD,'Kb')[0]

shade = Shades(SELF.circle\_image\_yELlow,coord)

SELF.LISTofShades.append(shade)

PYGAME.mixer.Sound.play(SELF.draw\_sound)

RETURN

IF SELF.chessEnded:

coord = SELF.c.lookFOR(SELF.BOARD,'K'+SELF.winner)[0]

shade = Shades(SELF.circle\_image\_green\_big,coord)

SELF.LISTofShades.append(shade)

IF SELF.winner=='w':

PYGAME.mixer.Sound.play(SELF.whitewin\_sound)

ELSE:

PYGAME.mixer.Sound.play(SELF.blackwin\_sound)

IF SELF.c.isCheck(SELF.position,'white'):

coord = SELF.c.lookFOR(SELF.BOARD,'Kw')[0]

shade = Shades(SELF.circle\_image\_red,coord)

SELF.LISTofShades.append(shade)

PYGAME.mixer.Sound.play(SELF.checkmate\_sound)

IF SELF.c.isCheck(SELF.position,'black'):

coord = SELF.c.lookFOR(SELF.BOARD,'Kb')[0]

shade = Shades(SELF.circle\_image\_red,coord)

SELF.LISTofShades.append(shade)

PYGAME.mixer.Sound.play(SELF.checkmate\_sound)

FOR pos in LISTofTuples:

IF SELF.c.isOccupied(SELF.BOARD,pos[0],pos[1]):

img = SELF.circle\_image\_capture

ELSE:

img = SELF.circle\_image\_green

shade = Shades(img,pos)

SELF.LISTofShades.append(shade)

DEF drawBOARD(SELF):

SELF.screen.blit(SELF.background,(0,0))

IF SELF.PLAYER==1:

order = [SELF.LISTofWhitePieces,SELF.LISTofBlackPieces]

ELSE:

order = [SELF.LISTofBlackPieces,SELF.LISTofWhitePieces]

IF SELF.isTransition:

order = LIST(reversed(order))

IF SELF.isDraw or SELF.chessEnded or SELF.isAIThink:

FOR shade in SELF.LISTofShades:

img,chess\_coord = shade.getInfo()

PIXEL\_coord = SELF.chess\_coord\_to\_PIXELs(chess\_coord)

SELF.screen.blit(img,PIXEL\_coord)

IF SELF.prevMove[0]!=-1 and NOT SELF.isTransition:

x,y,x2,y2 = SELF.prevMove

SELF.screen.blit(SELF.yELlowbox\_image,SELF.chess\_coord\_to\_PIXELs((x,y)))

SELF.screen.blit(SELF.yELlowbox\_image,SELF.chess\_coord\_to\_PIXELs((x2,y2)))

FOR piece in order[0]:

chess\_coord,subsection,pos = piece.getInfo()

PIXEL\_coord = SELF.chess\_coord\_to\_PIXELs(chess\_coord)

IF pos==(-1,-1):

SELF.screen.blit(SELF.pieces\_image,PIXEL\_coord,subsection)

ELSE:

SELF.screen.blit(SELF.pieces\_image,pos,subsection)

#Blit the shades in between:

IF NOT (SELF.isDraw or SELF.chessEnded or SELF.isAIThink):

FOR shade in SELF.LISTofShades:

img,chess\_coord = shade.getInfo()

PIXEL\_coord = SELF.chess\_coord\_to\_PIXELs(chess\_coord)

SELF.screen.blit(img,PIXEL\_coord)

#Potentially capturing pieces:

FOR piece in order[1]:

chess\_coord,subsection,pos = piece.getInfo()

PIXEL\_coord = SELF.chess\_coord\_to\_PIXELs(chess\_coord)

IF pos==(-1,-1):

SELF.screen.blit(SELF.pieces\_image,PIXEL\_coord,subsection)

ELSE:

SELF.screen.blit(SELF.pieces\_image,pos,subsection)

DEF media(SELF):

SELF.background = PYGAME.image.load('Media\\BOARD2.png').convert()

PIECES\_IMAGE = PYGAME.image.load('Media\\Chess\_Pieces\_Sprite.png').convert\_alpha()

CIRCLE\_IMAGE\_GREEN = PYGAME.image.load('Media\\green\_circle\_small.png').convert\_alpha()

CIRCLE\_IMAGE\_CAPTURE = PYGAME.image.load('Media\\green\_circle\_neg.png').convert\_alpha()

CIRCLE\_IMAGE\_RED = PYGAME.image.load('Media\\red\_circle\_big.png').convert\_alpha()

GREENBOX\_IMAGE = PYGAME.image.load('Media\\green\_box.png').convert\_alpha()

CIRCLE\_IMAGE\_YELLOW = PYGAME.image.load('Media\\yELlow\_circle\_big.png').convert\_alpha()

CIRCLE\_IMAGE\_GREEN\_BIG = PYGAME.image.load('Media\\green\_circle\_big.png').convert\_alpha()

YELLOWBOX\_IMAGE = PYGAME.image.load('Media\\yELlow\_box.png').convert\_alpha()

WITHFRIEND\_PIC = PYGAME.image.load('Media\\withfriend.png').convert\_alpha()

WITHAI\_PIC = PYGAME.image.load('Media\\withAI.png').convert\_alpha()

PLAYWHITE\_PIC = PYGAME.image.load('Media\\playWhite.png').convert\_alpha()

PLAYBLACK\_PIC = PYGAME.image.load('Media\\playBlack.png').convert\_alpha()

SELF.size\_of\_bg = SELF.background.get\_rect().size

SELF.square\_width = SELF.size\_of\_bg[0]//8

SELF.square\_height = SELF.size\_of\_bg[1]//8

SELF.pieces\_image = PYGAME.transFORm.scale(pieces\_image,

(SELF.square\_width\*6,SELF.square\_height\*2))

SELF.circle\_image\_green = PYGAME.transFORm.scale(circle\_image\_green,

(SELF.square\_width, SELF.square\_height))

SELF.circle\_image\_capture = PYGAME.transFORm.scale(circle\_image\_capture,

(SELF.square\_width, SELF.square\_height))

SELF.circle\_image\_red = PYGAME.transFORm.scale(circle\_image\_red,

(SELF.square\_width, SELF.square\_height))

SELF.greenbox\_image = PYGAME.transFORm.scale(greenbox\_image,

(SELF.square\_width, SELF.square\_height))

SELF.yELlowbox\_image = PYGAME.transFORm.scale(yELlowbox\_image,

(SELF.square\_width, SELF.square\_height))

SELF.circle\_image\_yELlow = PYGAME.transFORm.scale(circle\_image\_yELlow,

(SELF.square\_width, SELF.square\_height))

SELF.circle\_image\_green\_big = PYGAME.transFORm.scale(circle\_image\_green\_big,

(SELF.square\_width, SELF.square\_height))

SELF.withfriend\_pic = PYGAME.transFORm.scale(withfriend\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.withAI\_pic = PYGAME.transFORm.scale(withAI\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.playwhite\_pic = PYGAME.transFORm.scale(playwhite\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.playblack\_pic = PYGAME.transFORm.scale(playblack\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.flipEnabled\_pic = PYGAME.transFORm.scale(flipEnabled\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.flipDisabled\_pic = PYGAME.transFORm.scale(flipDisabled\_pic,

(SELF.square\_width\*4,SELF.square\_height\*4))

SELF.wELcome\_sound = PYGAME.mixer.Sound("VOICE\wELcome.wav")

SELF.exit\_sound = PYGAME.mixer.Sound("VOICE\exit.wav")

SELF.flip\_sound = PYGAME.mixer.Sound("VOICE\Flip.wav")

SELF.COLOR\_sound = PYGAME.mixer.Sound("VOICE\COLOR.wav")

SELF.thinking\_sound = PYGAME.mixer.Sound("VOICE\Thinking.wav")

SELF.dIFficulty\_sound=PYGAME.mixer.Sound("VOICE\dIFficulty.wav")

SELF.turn\_sound=PYGAME.mixer.Sound("VOICE\Turn.wav")

SELF.checkmate\_sound = PYGAME.mixer.Sound("VOICE\check.wav")

SELF.draw\_sound = PYGAME.mixer.Sound("VOICE\draw.wav")

SELF.whitewin\_sound = PYGAME.mixer.Sound("VOICE\whitewins.wav")

SELF.blackwin\_sound = PYGAME.mixer.Sound("VOICE\Blackwins.wav")

SELF.blackturn\_sound = PYGAME.mixer.Sound("VOICE\Blackturn.wav")

SELF.whiteturn\_sound = PYGAME.mixer.Sound("VOICE\whiteturn.wav")

SELF.piece\_sound=PYGAME.mixer.Sound("VOICE\piecehit.wav")

SELF.destination\_sound=PYGAME.mixer.Sound("VOICE\destination.wav")

SELF.instructions\_sound = PYGAME.mixer.Sound("VOICE\instructions.wav")

SELF.repeat\_sound = PYGAME.mixer.Sound("VOICE\Repeat.wav")

SELF.SELECTpiece\_sound = PYGAME.mixer.Sound("VOICE\SELECTpiece.wav")

SELF.requesterror\_sound = PYGAME.mixer.Sound("VOICE\Requesterror.wav")

SELF.control\_sound = PYGAME.mixer.Sound("VOICE\control.wav")

SELF.control\_sound=PYGAME.mixer.sound.play(self.invalid\_sound)

DEF initialize(SELF):

SELF.LISTofWhitePieces,SELF.LISTofBlackPieces = SELF.createPieces(SELF.BOARD)

SELF.LISTofShades = []

SELF.isDown = False #Variable that shows IF the mouse is being hELd down

SELF.isClicked = False #To keep track of whether a piece was clicked in order

SELF.isTransition = False #Keeps track of whether or NOT a piece is being animated.

SELF.isDraw = False #Will store True IF the game ended with a draw

SELF.chessEnded = False #Will become True once the chess game ends by checkmate, stalemate, etc.

SELF.isRecord = False #Set this to True IF you want to record moves to the Opening Book. Do NOT

SELF.isAIThink = False #Stores whether or NOT the AI is calculating the best move to be played.

SELF.openings = DEFaultdict(LIST)

TRY:

file\_handle = open('openingTable.txt','r')

SELF.openings = pickle.loads(file\_handle.read())

EXCEPT:

IF SELF.isRecord:

file\_handle = open('openingTable.txt','w')

SELF.letters\_dict = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, 'g': 6, 'h': 7}#dictionary FOR VOICE

SELF.numbers\_dict = {'1': 7, '2': 6, '3': 5, '4': 4, '5': 3, '6': 2, '7': 1, '8': 0}#dictionary FOR VOICE

SELF.piece\_SELECTed\_by\_VOICE=False

SELF.r = sr.Recognizer()#speechrecognition CLASS object

SELF.r.dynamic\_energy\_threshold = False

SELF.r.energy\_threshold = 400

SELF.searched = {}

SELF.prevMove = [-1,-1,-1,-1] #Also a global varible that stores the last move played, to

SELF.ax,SELF.ay=0,0

SELF.numm = 0

SELF.isMenu = True

SELF.isAI = -1

SELF.isFlip = -1

SELF.AIPLAYER = -1

SELF.gameEnded = False

DEF startMenu(SELF):

SELF.BOARDImage = PYGAME.image.load('newMedia/ChessImage.png')

SELF.BOARDImage = PYGAME.transFORm.scale(SELF.BOARDImage, (300, 300))

SELF.PLAYER1 = PYGAME.image.load('newMedia/play1.png')

SELF.PLAYER1 = PYGAME.transFORm.scale(SELF.PLAYER1, (280, 75))

SELF.PLAYER2 = PYGAME.image.load('newMedia/play2.png')

SELF.PLAYER2 = PYGAME.transFORm.scale(SELF.PLAYER2, (280, 75))

SELF.startPage.blit(SELF.box,(0,0))

SELF.startPage.blit(SELF.BOARDImage, (450-275, 60-15))

SELF.startPage.blit(SELF.PLAYER1, (460-275, 380-15))

SELF.startPage.blit(SELF.PLAYER2, (460-275, 470-15))

SELF.screen.blit(SELF.startPage, (0, 0))

DEF play1Menu\_A(SELF):

SELF.SELECTCOLOR = PYGAME.image.load('newMedia/SELECTCOLOR.png')

SELF.SELECTCOLOR = PYGAME.transFORm.scale(SELF.SELECTCOLOR, (350, 80))

SELF.playasblack = PYGAME.image.load('newMedia/playBlack.png')

SELF.playasblack = PYGAME.transFORm.scale(SELF.playasblack, (250, 250))

SELF.playaswhite = PYGAME.image.load('newMedia/playWhite.png')

SELF.playaswhite = PYGAME.transFORm.scale(SELF.playaswhite, (250, 250))

SELF.COLORPage.blit(SELF.box,(0,0))

SELF.COLORPage.blit(SELF.SELECTCOLOR, (425-275, 80-15))

SELF.COLORPage.blit(SELF.playasblack, (325-275, 280-15))

SELF.COLORPage.blit(SELF.playaswhite, (625-275, 280-15))

SELF.screen.blit(SELF.COLORPage, (0, 0))

GLOBAL play\_sound

IF play\_sound:

PLAY\_SOUND = False

PYGAME.mixer.Sound.play(SELF.COLOR\_sound)

DEF play1Menu\_B(SELF):

SELF.SELECTDIFficulty = PYGAME.image.load('newMedia/SELECTDIFficulty.png')

SELF.SELECTDIFficulty = PYGAME.transFORm.scale(SELF.SELECTDIFficulty, (350, 80))

SELF.Easy = PYGAME.image.load('newMedia/Easy.png')

SELF.Easy = PYGAME.transFORm.scale(SELF.Easy, (180, 180))

SELF.Medium = PYGAME.image.load('newMedia/Medium.png')

SELF.Medium = PYGAME.transFORm.scale(SELF.Medium, (180, 180))

SELF.Hard = PYGAME.image.load('newMedia/Hard.png')

SELF.Hard = PYGAME.transFORm.scale(SELF.Hard, (180, 180))

SELF.dIFfPage.blit(SELF.box,(0,0))

SELF.dIFfPage.blit(SELF.SELECTDIFficulty, (425-275, 80-15))

SELF.dIFfPage.blit(SELF.Easy, (309-275, 250-15))

SELF.dIFfPage.blit(SELF.Medium, (509-275, 250-15))

SELF.dIFfPage.blit(SELF.Hard, (709-275, 250-15))

SELF.screen.blit(SELF.dIFfPage, (0, 0))

SELF.dIFfMenu = 0

GLOBAL play\_sound

IF play\_sound:

PLAY\_SOUND = False

PYGAME.mixer.Sound.play(SELF.dIFficulty\_sound)

DEF play2Menu(SELF):

SELF.SELECTflip = PYGAME.image.load('newMedia/Flip.png')

SELF.SELECTflip = PYGAME.transFORm.scale(SELF.SELECTflip, (350, 80))

SELF.enableflip = PYGAME.image.load('newMedia/enableFlip.png')

SELF.enableflip = PYGAME.transFORm.scale(SELF.enableflip, (250, 250))

SELF.disableflip = PYGAME.image.load('newMedia/disableFlip.png')

SELF.disableflip = PYGAME.transFORm.scale(SELF.disableflip, (250, 250))

SELF.flipPage.blit(SELF.box,(0,0))

SELF.flipPage.blit(SELF.SELECTflip, (425-275, 80-15))

SELF.flipPage.blit(SELF.enableflip, (325-275, 280-15))

SELF.flipPage.blit(SELF.disableflip, (625-275, 280-15))

SELF.screen.blit(SELF.flipPage, (0, 0))

GLOBAL play\_sound

IF play\_sound:

PLAY\_SOUND = False

PYGAME.mixer.Sound.play(SELF.flip\_sound)

DEF SELECTMenu(SELF):

SELF.SELECTmode = PYGAME.image.load('newMedia/SELECTMode.png')

SELF.SELECTmode = PYGAME.transFORm.scale(SELF.SELECTmode, (350, 80))

SELF.bymouse = PYGAME.image.load('newMedia/controlMouse.png')

SELF.bymouse = PYGAME.transFORm.scale(SELF.bymouse, (250, 250))

SELF.byVOICE = PYGAME.image.load('newMedia/controlVOICE.png')

SELF.byVOICE = PYGAME.transFORm.scale(SELF.byVOICE, (250, 250))

SELF.SELECTPage.blit(SELF.box, (0, 0))

SELF.SELECTPage.blit(SELF.SELECTmode, (425 - 275, 80 - 15))

SELF.SELECTPage.blit(SELF.bymouse, (325 - 275, 280 - 15))

SELF.SELECTPage.blit(SELF.byVOICE, (625 - 275, 280 - 15))

SELF.screen.blit(SELF.SELECTPage, (0, 0))

GLOBAL play\_sound

IF play\_sound:

PLAY\_SOUND = False

PYGAME.mixer.Sound.play(SELF.control\_sound)

DEF call\_BOARD(SELF):

SELF.drawBOARD()

SELF.isMenu = False

IF SELF.isAI and SELF.AIPLAYER==0:

COLORsign=1

SELF.bestMoveRETURN = []

SELF.move\_thread = threading.Thread(target = SELF.a.negamax,

args = (SELF.position,SELF.levEL,-1000000,1000000,COLORsign,SELF.bestMoveRETURN,SELF.openings,SELF.searched))

SELF.move\_thread.start()

SELF.isAIThink = True

FOR event in pygame.event.get():

IF event.type == KEYDOWN:

IF event.key == K\_e:

SELF.gameEnded = True

ELIF event.type == QUIT:

SELF.gameEnded = True

PYGAME.mixer.Sound.play(self.exit\_sound)

BREAK

IF event.type == MOUSEBUTTONUP:

SELF.onClick()

DEF onClick(SELF):

global play\_sound

posx, posy = PYGAME.mouse.get\_pos()

IF SELF.buttons[1][0] < posx < SELF.buttons[1][0] + SELF.buttons[1][2]:

IF SELF.buttons[1][1] < posy < SELF.buttons[1][1] + SELF.buttons[1][3] and SELF.isAI == -1 :

SELF.isAI = True

posx , posy = (0 , 0)

IF SELF.buttons[2][0] < posx < SELF.buttons[2][0] + SELF.buttons[2][2] :

IF SELF.buttons[2][1] < posy < SELF.buttons[2][1] + SELF.buttons[2][3] and SELF.isAI == -1:

SELF.isAI = False

posx, posy = (0, 0)

IF SELF.buttons[3][0] < posx < SELF.buttons[3][0] + SELF.buttons[3][2]:

IF SELF.buttons[3][1] < posy < SELF.buttons[3][1] + SELF.buttons[3][3]:

IF SELF.isAI == True:

IF SELF.dIFfMenu == -1:

SELF.AIPLAYER = 0

SELF.isFlip = False

SELF.dIFfMenu = 1

posx, posy = (0, 0)

PLAY\_SOUND=True

ELIF SELF.isAI == True and SELF.SELECT == 1:

SELF.SELECT = 2

SELF.TEMP = 1

posx, posy = (0, 0)

PRINT("Mouse Operated")

ELIF SELF.isAI == False:

SELF.isFlip = True

SELF.TEMP = -1

posx, posy = (0, 0)

IF SELF.buttons[4][0] < posx < SELF.buttons[4][0] + SELF.buttons[4][2]:

IF SELF.buttons[4][1] < posy < SELF.buttons[4][1] + SELF.buttons[4][3]:

IF SELF.isAI == True:

IF SELF.dIFfMenu == -1:

SELF.AIPLAYER = 1

SELF.isFlip = False

SELF.dIFfMenu = 1

posx, posy = (0, 0)

play\_sound = True

ELIF SELF.isAI == True and SELF.SELECT == 1:

SELF.SELECT = 3

SELF.TEMP = 1

posx, posy = (0, 0)

PRINT("VOICE Operated")

ELIF SELF.isAI == False:

SELF.isFlip = False

SELF.TEMP = -1

posx, posy = (0, 0)

IF SELF.buttons[5][0] < posx < SELF.buttons[5][0] + SELF.buttons[5][2]:

IF SELF.buttons[5][1] < posy < SELF.buttons[5][1] + SELF.buttons[5][3]:

SELF.levEL = 1

SELF.SELECT = 1

posx, posy = (0, 0)

play\_sound = True

IF SELF.buttons[6][0] < posx < SELF.buttons[6][0] + SELF.buttons[6][2]:

IF SELF.buttons[6][1] < posy < SELF.buttons[6][1] + SELF.buttons[6][3]:

SELF.levEL = 2

SELF.SELECT = 1

posx, posy = (0, 0)

play\_sound = True

IF SELF.buttons[7][0] < posx < SELF.buttons[7][0] + SELF.buttons[7][2]:

IF SELF.buttons[7][1] < posy < SELF.buttons[7][1] + SELF.buttons[7][3]:

SELF.levEL = 3

SELF.SELECT = 1

posx, posy = (0, 0)

play\_sound = True

DEF Thinking(SELF):

SELF.ax+=1

IF SELF.ax==8:

SELF.ay+=1

SELF.ax=0

IF SELF.ay==8:

SELF.ax,SELF.ay=0,0

IF SELF.ax%4==0:

SELF.createShades([])

IF SELF.AIPLAYER==0:

SELF.LISTofShades.append(Shades(SELF.greenbox\_image,(7-SELF.ax,7-SELF.ay)))

ELSE:

SELF.LISTofShades.append(Shades(SELF.greenbox\_image,(SELF.ax,SELF.ay)))

GUI()

END

# **6. EXTERNAL INTERFACE REQUIREMENTS**

## 6.1 User Interface

Chess application shall provide a full color graphical user interface and take input from the mobile and voice.

Images for the chess pieces themselves shall be sourced from a royalty-free provider and follow a traditional theme, as in the example below.

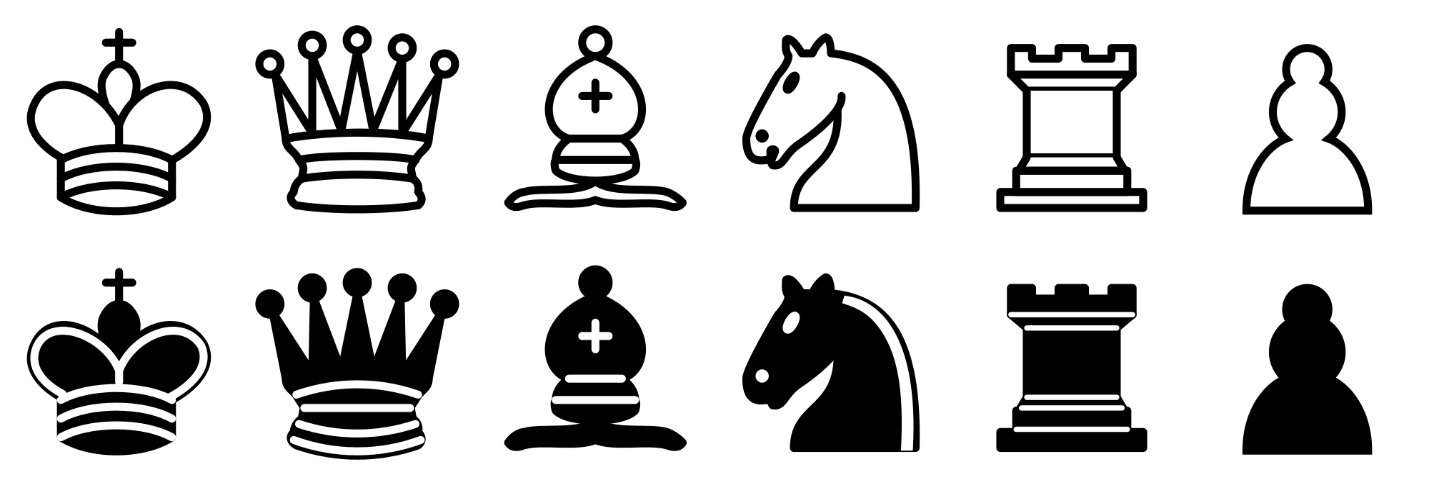


FIG.6.1. Pieces Images

**Pieces:** the above pieces are used to play the game. There are six pieces named King, Queen, Bishop, Knight, Rook, and pawn in black and white color.

FIG 6.2 Identification Images

**Select Piece**: The selected piece square is Identified by the dark green color.

**Possible ways**: the selected piece movements of possible ways are shown in the light green color. It is a dominant color in nature that makes you think of growth.

**King wins**: if the king wins it can be identified by the Green circle within the position of king.

**King lost**: If the king lost it can be identified by the Red circle within that square box.

**Possible Attack**: If any of the pieces attack other pieces it shows by the red color over the pieces.

**Match Draw**: it can be identified by the Yellow circle of both the Kings square box.

**Previous move:** previous moves are Identified by the Grey color square box.

**Identification Images:** the above images are used to make the difference between each move.

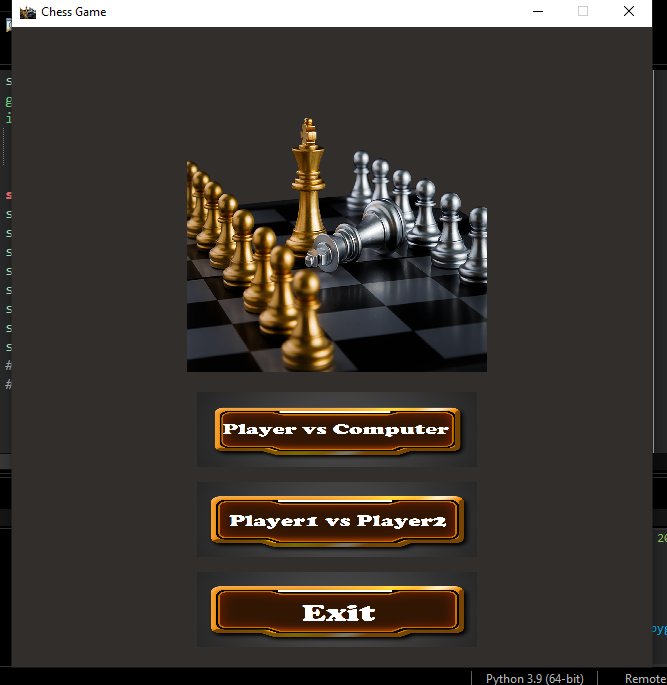


FIG.6.3. Menu page

**Menu Page**: From the little screen, chess after the choice of going to the starting that holds player1, player2 and exit by selecting on the appropriate button, similar to the one seen in this above sample design.

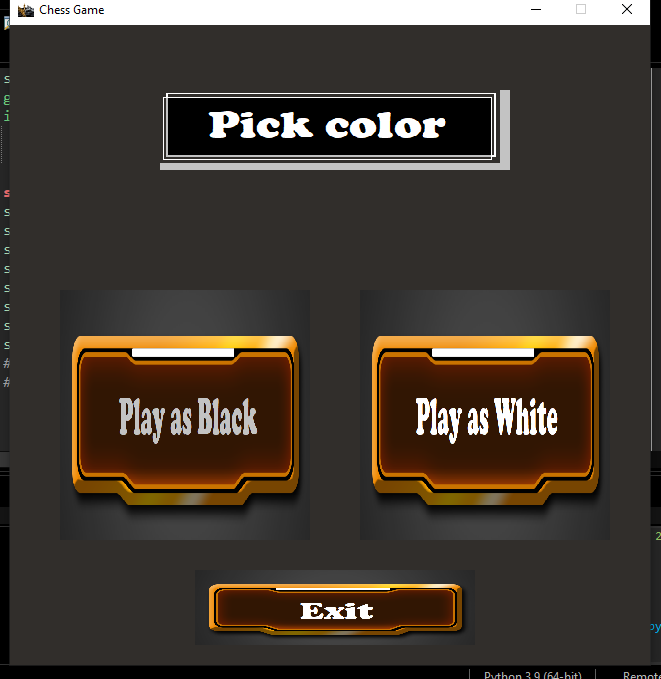


FIG.6.4. Color page

**Color page**: If the user selects the player1 button, it's moved to pick the color page. It has hold the play as black and play as white by selecting on the appropriate button, similar to the one seen in this above sample design.

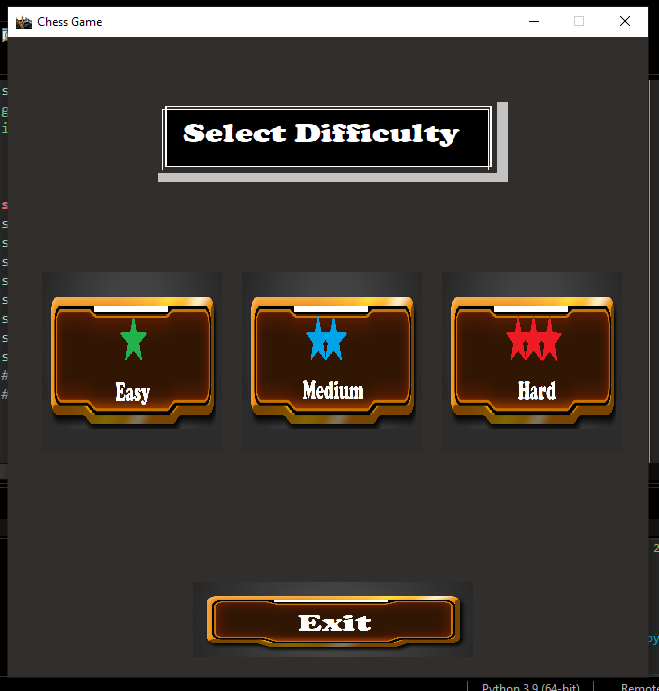


FIG.6.5. Select difficult page

**Select difficult page**: If user selects color then the selected difficult page is visible, It holds the easy, medium and hard by selecting the appropriate button, similar to the scene in this above sample page.

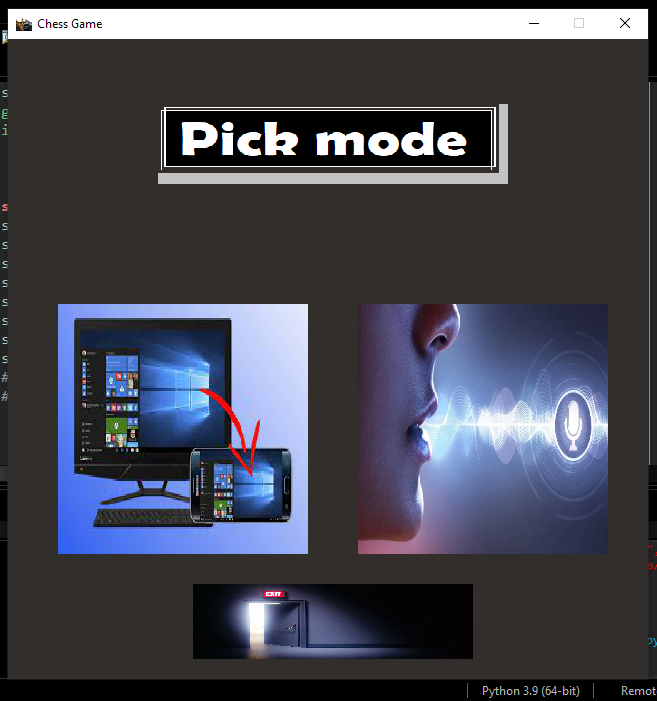


FIG.6.6. Control page

**Control page**: If the user selected difficult mode then the pick mode is visible, it holds the mobile control and voice control by selecting the appropriate button, similar to the one seen in the above sample page.

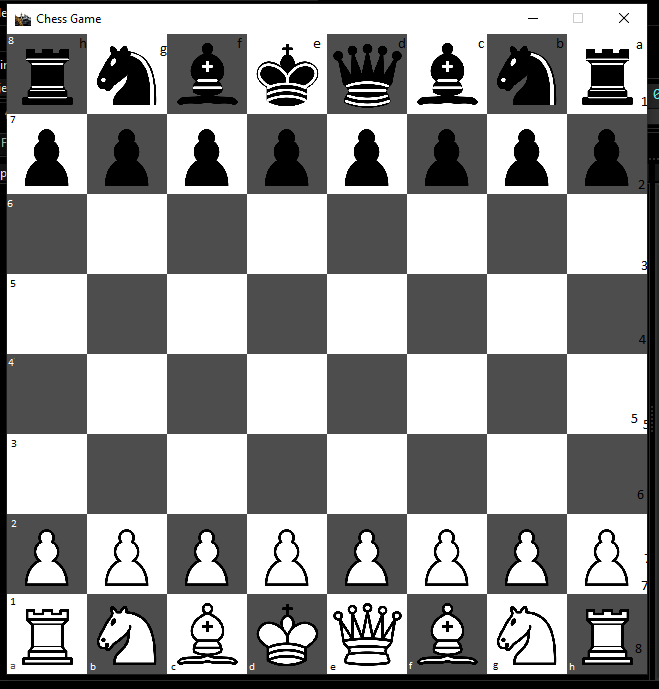


FIG.6.7. Game page

**Game page**: If the user selects the pick mode is visible. The sample chess board screen displays the sample image given above.

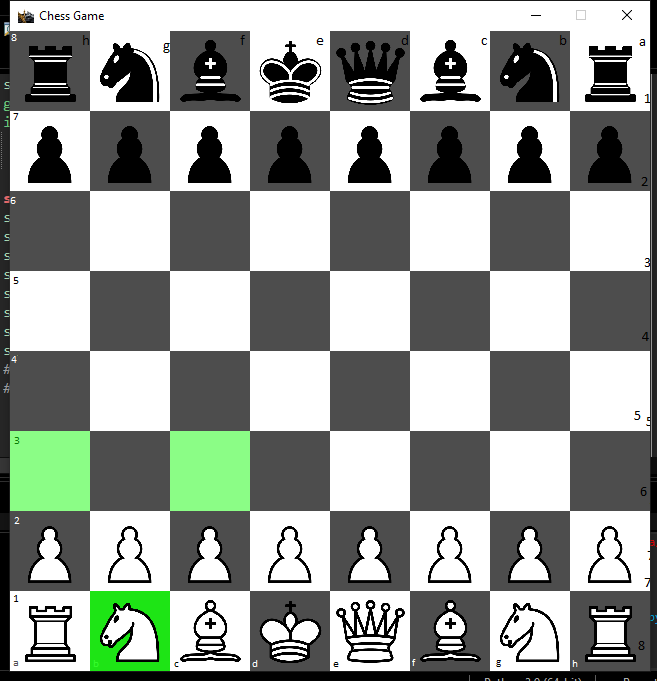


FIG.6.8. Move page

**Move page**: The above images show the possible ways of the knight.

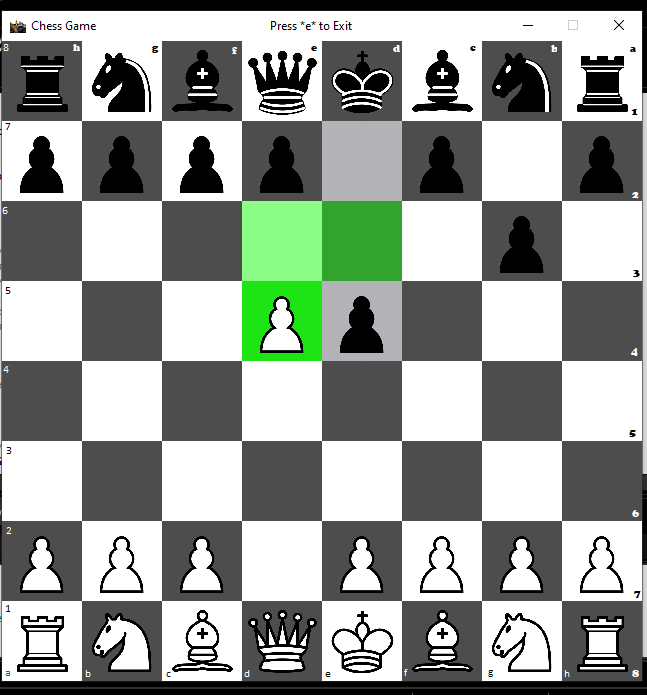


FIG.6.9.En-passant

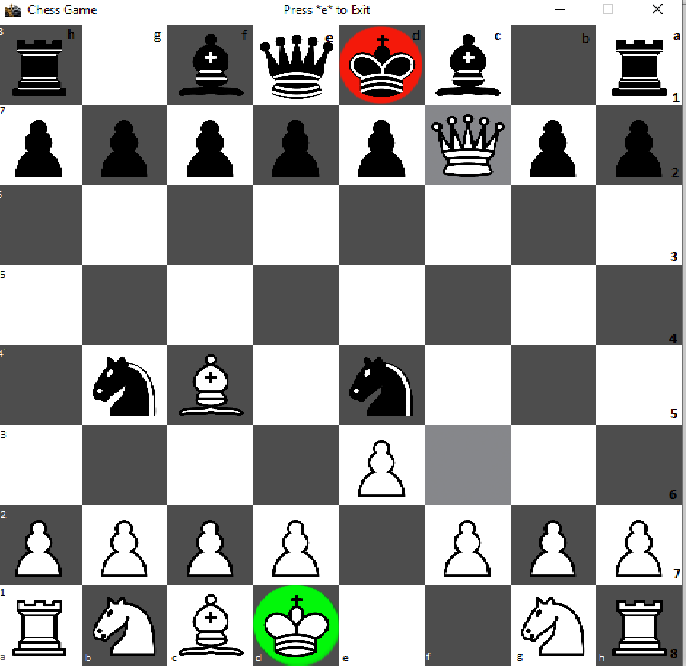


FIG.6.10.white wins

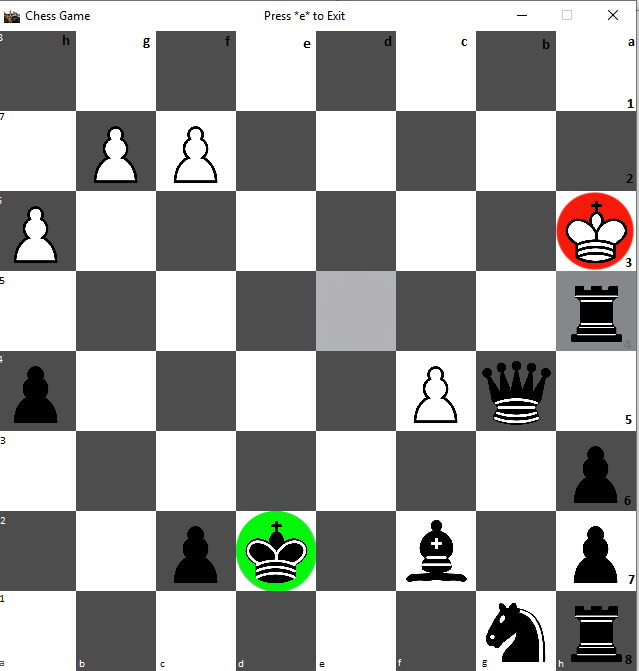


FIG.6.11.Black wins

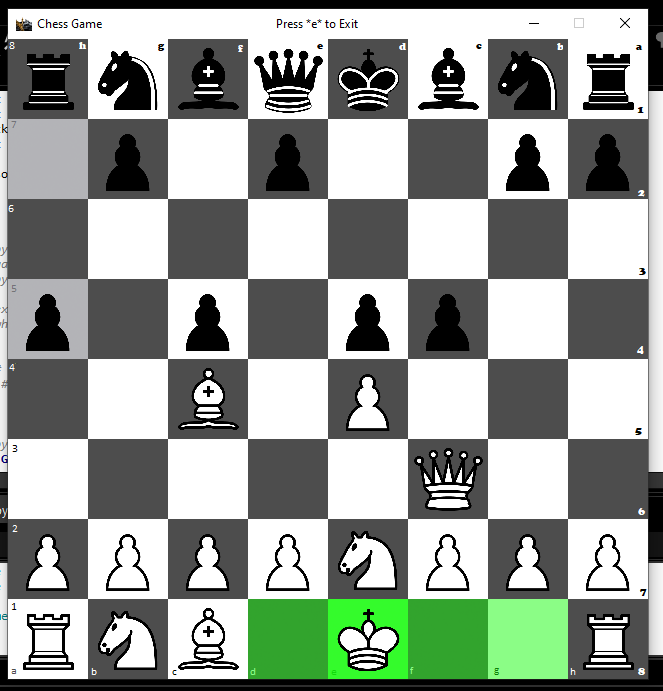


FIG.6.12.Castling rights

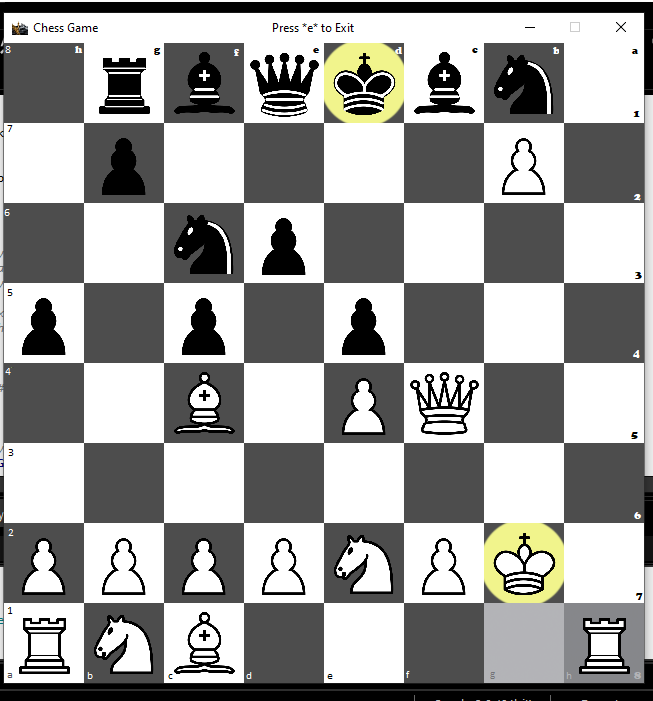


FIG.6.13. Match Draw

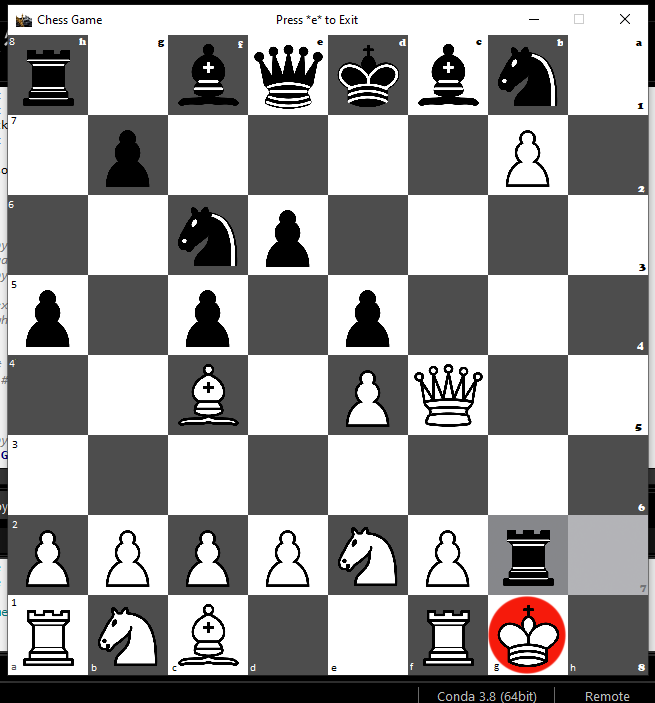


FIG.6.14.Check

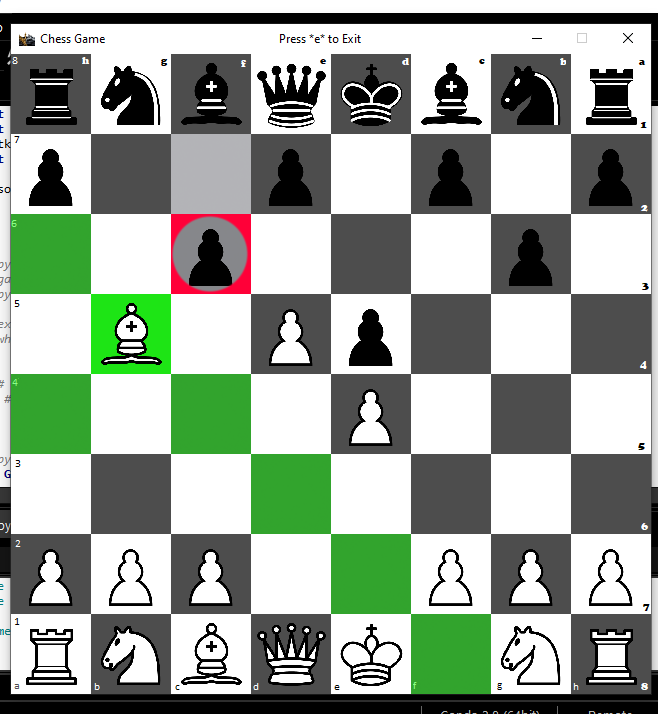


FIG.6.15.Piece clash

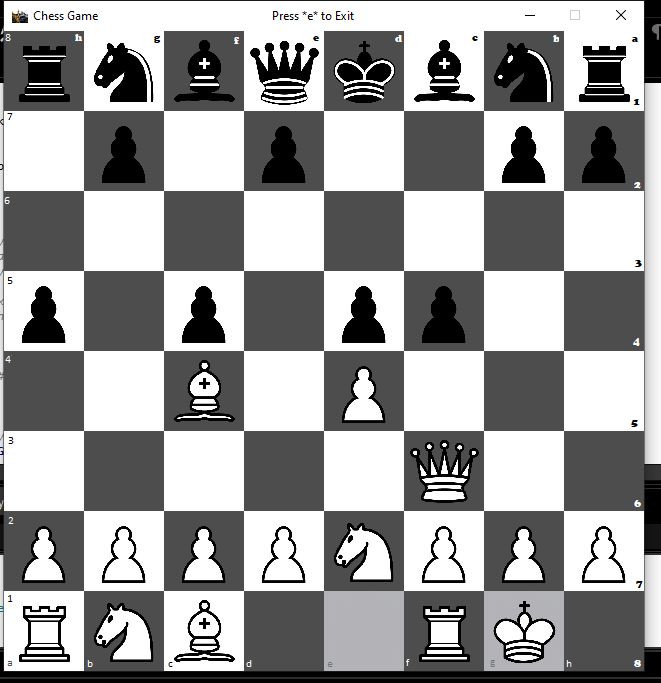


FIG.6.16. Previous move



FIG.6.17.Invalid move

# 7. LIMITATIONS:

1. Users must select the correct options which they want, otherwise the user cannot get back to the previous page.
2. Two players must use only voice or mobile input at the same time. Ex: one player mobile another player mic input is not valid.
3. When the pawn reaches its limit, the pawn is promoted as Queen piece only, other power pieces (Rook, Knight, Bishop) can't be promoted.
4. We can use the voice input while playing the game only, we can’t give the voice input for selecting the menu, color, difficulty, or control.
5. Players should use one chess board at a time for playing the game.
6. If both players are on the same wi-fi / USB cable can only play the game.
7. In two player mode (player1 vs player2), the top of the board has black pieces and the bottom of the board as white pieces. You can’t swap it.
8. At voice and mobile input algebraic notations are not visible in the game page.
9. Users can’t save and pause the game.
10. The player1 must select the white pieces, player2 select the black pieces. The users can decide who has to be player1 and player2.
11. If the player won the game, the player can’t start a new game on the same page. Players should close the window and run the game again.
12. Clashed pieces are not visible to the players.
13. If the player completed their turn, they can’t undo the last move.
14. Users can’t view the player name in the game page.

# 8. SCOPE FOR DEVELOPMENT

1. If the player completed their turn, they can undo the last move.
2. If a user selects incorrect options, they can get back to the previous page to select the correct options.
3. When the pawn reaches its limit, it can change as a power coin for the user 's wish.
4. Clashed pieces can be visible to the player.
5. The player's name should be visible in the game page.
6. Two players use different wifi to play the game.
7. Voice and mobile input algebraic notations are visible in the game page.
8. Users can save and pause the game.
9. The player won the game, then the player can start a new game on the same page.

# 9.SOURCE CODE:

# This program is a chess game. The user may play against a friend or the

# computer.

#

# The game state is mainly stored as a 2D list of strings, and most of the

# processing is thus done on a list of strings.

#

# The GUI takes the current state and displays it on the screen. The GUI allows

# drag and drop movement of pieces,click-click movement and voice movement.

#

# The AI that plays against the human evaluates all possible moves made by either

# player up to a certain level of depth. The AI evaluates each position by giving

# it a score. The higher the value of the score, the more favourable a position

# is for white and the lower the value of the score, the more favourable the

# position is for black. Knowing that white will try to get the score to be higher

# and black will try and get the score to be lower, the AI assumes best play from

# either side as it traverses up the search tree and chooses the best move to be

# played. A problem that may arise is the number of postions that need to be

# evaulated. Even at 3 levels of depth, thousands of positions have to be

# evaluatd.

# Several methods are used in this program to reduce positions that are searched:

# 1. Alpha-beta pruning: As a result of evaluating a position it can be found

# that a portion of the search tree can be ignored as no further evaluations can

# guarantee better results. This can happen because white and black area against

# one another. White plays what is best for it and black plays what is best for it,

# so it would make sense for white to ignore any portion of the tree where black

# has a clear upperhand that it can choose to play.

# 2. Transposition table: Often, two different pathways in a search tree can result

# in the same board being evaluated. Instead of evaluating the same board several

# times, the program stores a table of values in a dictionary where the keys are

# the positions. This way, repeated positions can have their evaluations looked up

# fairly quickly, as the board state is hashed.

# 3. Opening Book - The opening book is again a dictionary that stores board

# positions often seen in the beginning few moves in chess. Appropraite moves that

# can be played at such positions is stored in the dictionary. A random move is

# selected and played from the list of suggested moves wihtout searching if the AI

# finds itself confronting a such a board postion. Note that this opening book was

# recorded by myself and so it does not have many positions stored in it.

#

# In order to traverse the search tree as above, the AI needs to know how to evaluate the

# board at any position to decide if white or black has the advantage. My evaluation

# function currently looks at three main things when evaluating the board:

# a) Material for white and black. Each piece has a value and the more pieces you have,

# the better off your position is likely to be. For example, if white has an extra

# queen, it has an advantage over black.

# b) Piece-square table values - for each piece, there is a table that stores the best

# squares that the particular piece should occupy. So if white has a knight at a

# good square that controls the centre of the board, whereas black has a knight

# at the corner of the board, the situation is evaluated as being more favourable

# for white.

# c) Reduction in points for doubled pawns, isolated pawns, and blocked pawns. If any

# side has a set of pawns with the above features their points are slightly lower

# to indicate a slight disadvantage in such a position.

# d) A checkmate: a position where this has occured gets a very high point, so that the

# AI moves towards this if it can. (or avoids it).

#Import dependencies:

import pygame #Game library

from pygame.locals import \* #For useful variables..for eg QUIT

import copy #Library used to make exact copies of lists.

import pickle #Library used to store dictionaries in a text file and read them from text files.

import random #Used for making random selections

from collections import defaultdict #Used for giving dictionary values default data types.

from collections import Counter #For counting elements in a list effieciently.

import threading #To allow for AI to think simultaneously while the GUI is coloring the board.

import time

import speech\_recognition as sr

from tkinter import\*

import tkinter.messagebox

play\_sound=True

class GamePosition:

def \_\_init\_\_(self,board,player,castling\_rights,EnP\_Target,HMC,history = {}):

# Making an object of Commands Class

self.c=Commands()

# A 2D array containing information about piece postitions. Check main function to see an example of such

# a representation.

self.board = board

# Stores 0 or 1. If white to play, equals 0. If black to play, stores 1.

self.player = player

# A list that contains castling rights for white and black. Each castling right is a list that contains right

# to castle kingside and queenside.

# Stores the coordinates of a square that can be targeted by en passant capture.

self.EnP = EnP\_Target

self.castling = castling\_rights

# Half move clock. Stores the number of irreversible moves made so far, in order to help

# detect draw by 50 moves without any capture or pawn movement.

self.HMC = HMC

# A dictionary that stores as key a position (hashed) and the value of each of

# these keys represents the number of times each of these positions was repeated in order for this

# position to be reached.

self.history = history

def getboard(self):

return self.board

def setboard(self,board):

self.board = board

def getplayer(self):

return self.player

def setplayer(self,player):

self.player = player

def getCastleRights(self):

return self.castling

def setCastleRights(self,castling\_rights):

self.castling = castling\_rights

def getEnP(self):

return self.EnP

def setEnP(self, EnP\_Target):

self.EnP = EnP\_Target

def getHMC(self):

return self.HMC

def setHMC(self,HMC):

self.HMC = HMC

def checkRepition(self):

# Returns True if any of of the values in the history dictionary is greater than 3.

# This would mean a position had been repeated at least thrice in order to reach the

# current position in this game.

return any(value>=3 for value in self.history.values())

def addtoHistory(self,position):

# Generate a unique key out of the current position:

key = self.c.pos2key(position)

#Add it to the history dictionary.

self.history[key] = self.history.get(key,0) + 1

def gethistory(self):

return self.history

def clone(self):

# This method returns another instance of the current object with exactly the same

# parameters but independent of the current object.

clone = GamePosition(copy.deepcopy(self.board), # Independent copy

self.player,

copy.deepcopy(self.castling), # Independent copy

self.EnP,

self.HMC)

return clone

class Shades:

"""

It is used to shade the board

"""

def \_\_init\_\_(self,image,coord):

self.image = image

self.pos = coord

def getInfo(self):

return [self.image,self.pos]

class Piece:

"""

Piece clips the sprite into pieces to display on the board

"""

def \_\_init\_\_(self,pieceinfo,chess\_coord,square\_width, square\_height):

# pieceinfo is a string such as 'Qb'. The Q represents Queen and b

# shows the fact that it is black:

piece = pieceinfo[0]

color = pieceinfo[1]

# Get the information about where the image for this piece is stored

# on the overall sprite image with all the pieces. Note that

# square\_width and square\_height represent the size of a square on the

# chess board.

if piece=='K':

index = 0

elif piece=='Q':

index = 1

elif piece=='B':

index = 2

elif piece == 'N':

index = 3

elif piece == 'R':

index = 4

elif piece == 'P':

index = 5

left\_x =square\_width\*index

if color == 'w':

left\_y = 0

else:

left\_y = square\_height

self.pieceinfo = pieceinfo

# subsection defines the part of the sprite image that represents our

# piece

self.subsection = (left\_x,left\_y,square\_width,square\_height)

# There are two ways that the position of a piece is defined on the

# board.

# The default one used is the chess\_coord, which stores something

# like (3,2). It represents the chess coordinate where our piece image should

# be blitted. On the other hand, is pos does not hold the default value

# of (-1,-1), it will hold pixel coordinates such as (420,360) that represents

# the location in the window that the piece should be blitted on. This is

# useful for example if our piece is transitioning from a square to another:

self.chess\_coord = chess\_coord

self.pos = (-1,-1)

def getInfo(self):

return [self.chess\_coord, self.subsection,self.pos]

def setpos(self,pos):

self.pos = pos

def getpos(self):

return self.pos

def setcoord(self,coord):

self.chess\_coord = coord

class Commands:

"""

this class works with variables that hold the information about game state.

it contains chess processing functions

"""

def isOccupied(self,board,x,y):

# isOccupied(board,x,y) - Returns true if a given coordinate on the board is not empty, and

# false otherwise.

if board[int(y)][int(x)] == 0:

# The square has nothing on it.

return False

return True

def isOccupiedby(self,board,x,y,color):

# isOccupiedby(board,x,y,color) - Same as above, but only returns true if the square

# specified by the coordinates is of the specific color inputted.

if board[y][x] == 0:

# the square has nothing on it.

return False

if board[y][x][1] == color[0]:

# The square has a piece of the color inputted.

return True

# The square has a piece of the opposite color.

return False

def filterbyColor(self,board,listofTuples,color):

# filterbyColor(board,listofTuples,color) - This function takes the board state, a list

# of coordinates, and a color as input. It will return the same list, but without

# coordinates that are out of bounds of the board and also without those occupied by the

# pieces of the particular color passed to this function as an argument. In other words,

# if 'white' is passed in, it will not return any white occupied square.

filtered\_list = []

# Go through each coordinate:

for pos in listofTuples:

x = pos[0]

y = pos[1]

if x>=0 and x<=7 and y>=0 and y<=7 and not self.isOccupiedby(board,x,y,color):

# coordinates are on-board and no same-color piece is on the square.

filtered\_list.append(pos)

return filtered\_list

def lookfor(self,board,piece):

# lookfor(board,piece) - This functions takes the 2D array that represents a board and finds

# the indices of all the locations that is occupied by the specified piece. The list of

# indices is returned.

listofLocations = []

for row in range(8):

for col in range(8):

if board[row][col] == piece:

x = col

y = row

listofLocations.append((x,y))

return listofLocations

def isAttackedby(self,position,target\_x,target\_y,color):

# isAttackedby(position,target\_x,target\_y,color) - This function checks if the square specified

# by (target\_x,target\_y) coordinates is being attacked by any of a specific colored set of pieces.

# Get board

board = position.getboard()

# Get b from black or w from white

color = color[0]

# Get all the squares that are attacked by the particular side:

listofAttackedSquares = []

for x in range(8):

for y in range(8):

if board[y][x]!=0 and board[y][x][1]==color:

listofAttackedSquares.extend(

self.findPossibleSquares(position,x,y,True)) #The true argument

# prevents infinite recursion.

# Check if the target square falls under the range of attack by the specified

# side, and return it:

return (target\_x,target\_y) in listofAttackedSquares

def findPossibleSquares(self,position,x,y,AttackSearch=False):

# findPossibleSquares(position,x,y,AttackSearch=False) - This function takes as its input the

# current state of the chessboard, and a particular x and y coordinate. It will return for the

# piece on that board a list of possible coordinates it could move to, including captures and

# excluding illegal moves (eg moves that leave a king under check). AtttackSearch is an

# argument used to ensure infinite recursions do not occur.

# Get individual component data from the position object:

board = position.getboard()

player = position.getplayer()

castling\_rights = position.getCastleRights()

EnP\_Target = position.getEnP()

# In case something goes wrong:

piece = board[y][x][0] #Pawn, rook, etc.

color = board[y][x][1] #w or b.

# Have the complimentary color stored for convenience:

enemy\_color = self.opp(color)

listofTuples = [] #Holds list of attacked squares.

if piece == 'P': #The piece is a pawn.

if color=='w': #The piece is white

if not self.isOccupied(board,x,y-1) and not AttackSearch:

#The piece immediately above is not occupied, append it.

listofTuples.append((x,y-1))

if y == 6 and not self.isOccupied(board,x,y-2):

#If pawn is at its initial position, it can move two squares.

listofTuples.append((x,y-2))

if x!=0 and self.isOccupiedby(board,x-1,y-1,'black'):

#The piece diagonally up and left of this pawn is a black piece.

#Also, this is not an 'a' file pawn (left edge pawn)

listofTuples.append((x-1,y-1))

if x!=7 and self.isOccupiedby(board,x+1,y-1,'black'):

#The piece diagonally up and right of this pawn is a black one.

#Also, this is not an 'h' file pawn.

listofTuples.append((x+1,y-1))

if EnP\_Target!=-1: #There is a possible en pasant target:

if EnP\_Target == (x-1,y-1) or EnP\_Target == (x+1,y-1):

#We're at the correct location to potentially perform en

#passant:

listofTuples.append(EnP\_Target)

elif color=='b': #The piece is black, same as above but opposite side.

if not self.isOccupied(board,x,y+1) and not AttackSearch:

listofTuples.append((x,y+1))

if y == 1 and not self.isOccupied(board,x,y+2):

listofTuples.append((x,y+2))

if x!=0 and self.isOccupiedby(board,x-1,y+1,'white'):

listofTuples.append((x-1,y+1))

if x!=7 and self.isOccupiedby(board,x+1,y+1,'white'):

listofTuples.append((x+1,y+1))

if EnP\_Target == (x-1,y+1) or EnP\_Target == (x+1,y+1):

listofTuples.append(EnP\_Target)

elif piece == 'R': #The piece is a rook.

#Get all the horizontal squares:

for i in [-1,1]:

#i is -1 then +1. This allows for searching right and left.

kx = x #This variable stores the x coordinate being looked at.

while True: #loop till break.

kx = kx + i #Searching left or right

if kx<=7 and kx>=0: #Making sure we're still in board.

if not self.isOccupied(board,kx,y):

#The square being looked at it empty. Our rook can move

#here.

listofTuples.append((kx,y))

else:

#The sqaure being looked at is occupied. If an enemy

#piece is occupying it, it can be captured so its a valid

#move.

if self.isOccupiedby(board,kx,y,enemy\_color):

listofTuples.append((kx,y))

#Regardless of the occupying piece color, the rook cannot

#jump over. No point continuing search beyond in this

#direction:

break

else: #We have exceeded the limits of the board

break

#Now using the same method, get the vertical squares:

for i in [-1,1]:

ky = y

while True:

ky = ky + i

if ky<=7 and ky>=0:

if not self.isOccupied(board,x,ky):

listofTuples.append((x,ky))

else:

if self.isOccupiedby(board,x,ky,enemy\_color):

listofTuples.append((x,ky))

break

else:

break

elif piece == 'N': #The piece is a knight.

#The knight can jump across a board. It can jump either two or one

#squares in the x or y direction, but must jump the complimentary amount

#in the other. In other words, if it jumps 2 sqaures in the x direction,

#it must jump one square in the y direction and vice versa.

for dx in [-2,-1,1,2]:

if abs(dx)==1:

sy = 2

else:

sy = 1

for dy in [-sy,+sy]:

listofTuples.append((x+dx,y+dy))

#Filter the list of tuples so that only valid squares exist.

listofTuples = self.filterbyColor(board,listofTuples,color)

elif piece == 'B': # A bishop.

#A bishop moves diagonally. This means a change in x is accompanied by a

#change in y-coordiante when the piece moves. The changes are exactly the

#same in magnitude and direction.

for dx in [-1,1]: #Allow two directions in x.

for dy in [-1,1]: #Similarly, up and down for y.

kx = x #These varibales store the coordinates of the square being

#observed.

ky = y

while True: #loop till broken.

kx = kx + dx #change x

ky = ky + dy #change y

if kx<=7 and kx>=0 and ky<=7 and ky>=0:

#square is on the board

if not self.isOccupied(board,kx,ky):

#The square is empty, so our bishop can go there.

listofTuples.append((kx,ky))

else:

#The square is not empty. If it has a piece of the

#enemy,our bishop can capture it:

if self.isOccupiedby(board,kx,ky,enemy\_color):

listofTuples.append((kx,ky))

#Bishops cannot jump over other pieces so terminate

#the search here:

break

else:

#Square is not on board. Stop looking for more in this

#direction:

break

elif piece == 'Q': #A queen

#A queen's possible targets are the union of all targets that a rook and

#a bishop could have made from the same location

#Temporarily pretend there is a rook on the spot:

board[y][x] = 'R' + color

list\_rook = self.findPossibleSquares(position,x,y,True)

#Temporarily pretend there is a bishop:

board[y][x] = 'B' + color

list\_bishop = self.findPossibleSquares(position,x,y,True)

#Merge the lists:

listofTuples = list\_rook + list\_bishop

#Change the piece back to a queen:

board[y][x] = 'Q' + color

elif piece == 'K': # A king!

#A king can make one step in any direction:

for dx in [-1,0,1]:

for dy in [-1,0,1]:

listofTuples.append((x+dx,y+dy))

#Make sure the targets aren't our own piece or off-board:

listofTuples = self.filterbyColor(board,listofTuples,color)

if not AttackSearch:

#Kings can potentially castle:

right = castling\_rights[player]

#Kingside

if (right[0] and #has right to castle

board[y][7]!=0 and #The rook square is not empty

board[y][7][0]=='R' and #There is a rook at the appropriate place

not self.isOccupied(board,x+1,y) and #The square on its right is empty

not self.isOccupied(board,x+2,y) and #The second square beyond is also empty

not self.isAttackedby(position,x,y,enemy\_color) and #The king isn't under atack

not self.isAttackedby(position,x+1,y,enemy\_color) and #Or the path through which

not self.isAttackedby(position,x+2,y,enemy\_color)):#it will move

listofTuples.append((x+2,y))

#Queenside

if (right[1] and #has right to castle

board[y][0]!=0 and #The rook square is not empty

board[y][0][0]=='R' and #The rook square is not empty

not self.isOccupied(board,x-1,y)and #The square on its left is empty

not self.isOccupied(board,x-2,y)and #The second square beyond is also empty

not self.isOccupied(board,x-3,y) and #And the one beyond.

not self.isAttackedby(position,x,y,enemy\_color) and #The king isn't under atack

not self.isAttackedby(position,x-1,y,enemy\_color) and #Or the path through which

not self.isAttackedby(position,x-2,y,enemy\_color)):#it will move

listofTuples.append((x-2,y)) #Let castling be an option.

# Make sure the king is not under attack as a result of this move:

if not AttackSearch:

new\_list = []

for tupleq in listofTuples:

x2 = int(tupleq[0])

y2 = int(tupleq[1])

temp\_pos = position.clone()

self.makemove(temp\_pos,x,y,x2,y2)

if not self.isCheck(temp\_pos,color):

new\_list.append(tupleq)

listofTuples = new\_list

return listofTuples

def makemove(self,position,x,y,x2,y2):

# makemove(position,x,y,x2,y2) - This function makes a move on the board. The position object

# gets updated here with new information. (x,y) are coordinates of the piece to be moved, and

# (x2,y2) are coordinates of the destination. (x2,y2) being correct destination (ie the move

# a valid one) is not checked for and is assumed to be the case.

# Get data from the position:

x = int(x)

y = int(y)

x2 = int(x2)

y2 = int(y2)

board = position.getboard()

piece = board[y][x]

if piece==0:

return

piece=piece[0]

color = board[y][x][1]

#Get the individual game components:

player = position.getplayer()

castling\_rights = position.getCastleRights()

EnP\_Target = position.getEnP()

half\_move\_clock = position.getHMC()

#Update the half move clock:

if self.isOccupied(board,x2,y2) or piece=='P':

#Either a capture was made or a pawn has moved:

half\_move\_clock = 0

else:

#An irreversible move was played:

half\_move\_clock += 1

#Make the move:

board[y2][x2] = board[y][x]

board[y][x] = 0

#Special piece requirements:

#King:

if piece == 'K':

#Ensure that since a King is moved, the castling

#rights are lost:

castling\_rights[player] = [False,False]

#If castling occured, place the rook at the appropriate location:

if abs(x2-x) == 2:

if color=='w':

l = 7

else:

l = 0

if x2>x:

board[l][5] = 'R'+color

board[l][7] = 0

else:

board[l][3] = 'R'+color

board[l][0] = 0

#Rook:

if piece=='R':

#The rook moved. Castling right for this rook must be removed.

if x==0 and y==0:

#Black queenside

castling\_rights[1][1] = False

elif x==7 and y==0:

#Black kingside

castling\_rights[1][0] = False

elif x==0 and y==7:

#White queenside

castling\_rights[0][1] = False

elif x==7 and y==7:

#White kingside

castling\_rights[0][0] = False

#Pawn:

if piece == 'P':

#If an en passant kill was made, the target enemy must die:

if EnP\_Target == (x2,y2):

if color=='w':

board[y2+1][x2] = 0

else:

board[y2-1][x2] = 0

#If a pawn moved two steps, there is a potential en passant

#target. Otherise, there isn't. Update the variable:

if abs(y2-y)==2:

EnP\_Target = (x,(y+y2)/2)

else:

EnP\_Target = -1

#If a pawn moves towards the end of the board, it needs to

#be promoted. Note that in this game a pawn is being promoted

#to a queen regardless of user choice.

if y2==0:

board[y2][x2] = 'Qw'

elif y2 == 7:

board[y2][x2] = 'Qb'

else:

#If a pawn did not move, the en passsant target is gone as well,

#since a turn has passed:

EnP\_Target = -1

#Since a move has been made, the other player

#should be the 'side to move'

player = 1 - player

#Update the position data:

position.setplayer(player)

position.setCastleRights(castling\_rights)

position.setEnP(EnP\_Target)

position.setHMC(half\_move\_clock)

def opp(self,color):

# opp(color) - Returns the complimentary color to the one passed. So inputting 'black' returns

# 'w', for example.

color = color[0]

if color == 'w':

oppcolor = 'b'

else:

oppcolor = 'w'

return oppcolor

def isCheck(self,position,color):

# isCheck(position,color) - This function takes a position as its input and checks if the

# King of the specified color is under attack by the enemy. Returns true if that is the case,

# and false otherwise.

#Get data:

board = position.getboard()

color = color[0]

enemy = self.opp(color)

piece = 'K' + color

#Get the coordinates of the king:

x,y = self.lookfor(board,piece)[0]

#Check if the position of the king is attacked by

#the enemy and return the result:

return self.isAttackedby(position,x,y,enemy)

def isCheckmate(self,position,color=-1):

# isCheckmate(position,color=-1) - This function tells you if a position is a checkmate.

# Color is an optional argument that may be passed to specifically check for mate against a

# specific color.

if color==-1:

return self.isCheckmate(position,'white') or self.isCheckmate(position,'b')

color = color[0]

if self.isCheck(position,color) and self.allMoves(position,color)==[]:

#The king casis under attack, and there are no possible moves for this side to make:

return True

#Either the king is not under attack or there are possible moves to be played:

return False

def isStalemate(self,position):

# isStalemate(position) - This function checks if a particular position is a stalemate.

# If it is, it returns true, otherwise it returns false.

#Get player to move:

player = position.getplayer()

#Get color:

if player==0:

color = 'w'

else:

color = 'b'

if not self.isCheck(position,color) and self.allMoves(position,color)==[]:

#The player to move is not under check yet cannot make a move.

#It is a stalemate.

return True

return False

def getallpieces(self,position,color):

# getallpieces(position,color) - This function returns a list of positions of all the pieces on

# the board of a particular color.

#Get the board:

board = position.getboard()

listofpos = []

for j in range(8):

for i in range(8):

if self.isOccupiedby(board,i,j,color):

listofpos.append((i,j))

return listofpos

def allMoves(self,position, color):

# allMoves(position, color) - This function takes as its argument a position and a color/colorsign

# that represents a side. It generates a list of all possible moves for that side and returns it.

#Find if it is white to play or black:

if color==1:

color = 'white'

elif color ==-1:

color = 'black'

color = color[0]

#Get all pieces controlled by this side:

listofpieces = self.getallpieces(position,color)

moves = []

#Loop through each piece:

for pos in listofpieces:

#For each piece, find all the targets it can attack:

targets = self.findPossibleSquares(position,pos[0],pos[1])

for target in targets:

#Save them all as possible moves:

moves.append([pos,target])

return moves

def pos2key(self,position):

# pos2key(position) - This function takes a position as input argument. For this particular # position, it will generate a unique key that can be used in a dictionary by making it hashable.

#Get board:

board = position.getboard()

#Convert the board into a tuple so it is hashable:

boardTuple = []

for row in board:

boardTuple.append(tuple(row))

boardTuple = tuple(boardTuple)

#Get castling rights:

rights = position.getCastleRights()

#Convert to a tuple:

tuplerights = (tuple(rights[0]),tuple(rights[1]))

#Generate the key, which is a tuple that also takes into account the side to play:

key = (boardTuple,position.getplayer(),

tuplerights)

#Return the key:

return key

###########################////////AI RELATED FUNCTIONS\\\\\\\\\\############################

class AI:

def \_\_init\_\_(self):

self.c=Commands()

def negamax( self,position,depth,alpha,beta,colorsign,bestMoveReturn,openings,searched,root=True):

#First check if the position is already stored in the opening database dictionary:

if root:

#Generate key from current position:

key = self.c.pos2key(position)

if key in openings:

#Return the best move to be played:

bestMoveReturn[:] = random.choice(openings[key])

return

#Access global variable that will store scores of positions already evaluated:

#If the depth is zero, we are at a leaf node (no more depth to be analysed):

if depth==0:

return colorsign\*self.evaluate(position)

#Generate all the moves that can be played:

moves = self.c.allMoves(position, colorsign)

#If there are no moves to be played, just evaluate the position and return it:

if moves==[]:

return colorsign\*self.evaluate(position)

#Initialize a best move for the root node:

if root:

bestMove = moves[0]

#Initialize the best move's value:

bestValue = -100000

#Go through each move:

for move in moves:

#Make a clone of the current move and perform the move on it:

newpos = position.clone()

self.c.makemove(newpos,move[0][0],move[0][1],move[1][0],move[1][1])

#Generate the key for the new resulting position:

key = self.c.pos2key(newpos)

#If this position was already searched before, retrieve its node value.

#Otherwise, calculate its node value and store it in the dictionary:

if key in searched:

value = searched[key]

else:

value = -self.negamax(newpos,depth-1, -beta,-alpha,-colorsign,[],openings,searched,False)

searched[key] = value

#If this move is better than the best so far:

if value>bestValue:

#Store it

bestValue = value

#If we're at root node, store the move as the best move:

if root:

bestMove = move

#Update the lower bound for this node:

alpha = max(alpha,value)

if alpha>=beta:

#If our lower bound is higher than the upper bound for this node, there

#is no need to look at further moves:

break

#If this is the root node, return the best move:

if root:

searched = {}

bestMoveReturn[:] = bestMove

return

#Otherwise, return the bestValue (i.e. value for this node.)

return bestValue

def evaluate(self,position):

if self.c.isCheckmate(position,'white'):

#Major advantage to black

return -20000

if self.c.isCheckmate(position,'black'):

#Major advantage to white

return 20000

#Get the board:

board = position.getboard()

#Flatten the board to a 1D array for faster calculations:

flatboard = [x for row in board for x in row]

#Create a counter object to count number of each pieces:

c = Counter(flatboard)

Qw = c['Qw']

Qb = c['Qb']

Rw = c['Rw']

Rb = c['Rb']

Bw = c['Bw']

Bb = c['Bb']

Nw = c['Nw']

Nb = c['Nb']

Pw = c['Pw']

Pb = c['Pb']

#Note: The above choices to flatten the board and to use a library

#to count pieces were attempts at making the AI more efficient.

#Perhaps using a 1D board throughout the entire program is one way

#to make the code more efficient.

#Calculate amount of material on both sides and the number of moves

#played so far in order to determine game phase:

whiteMaterial = 9\*Qw + 5\*Rw + 3\*Nw + 3\*Bw + 1\*Pw

blackMaterial = 9\*Qb + 5\*Rb + 3\*Nb + 3\*Bb + 1\*Pb

numofmoves = len(position.gethistory())

gamephase = 'opening'

if numofmoves>40 or (whiteMaterial<14 and blackMaterial<14):

gamephase = 'ending'

#A note again: Determining game phase is again one the attempts

#to make the AI smarter when analysing boards and has not been

#implemented to its full potential.

#Calculate number of doubled, blocked, and isolated pawns for

#both sides:

Dw = self.doubledPawns(board,'white')

Db = self.doubledPawns(board,'black')

Sw = self.blockedPawns(board,'white')

Sb = self.blockedPawns(board,'black')

Iw = self.isolatedPawns(board,'white')

Ib = self.isolatedPawns(board,'black')

#Evaluate position based on above data:

evaluation1 = 900\*(Qw - Qb) + 500\*(Rw - Rb) +330\*(Bw-Bb

)+320\*(Nw - Nb) +100\*(Pw - Pb) +-30\*(Dw-Db + Sw-Sb + Iw- Ib

)

#Evaluate position based on piece square tables:

evaluation2 = self.pieceSquareTable(flatboard,gamephase)

#Sum the evaluations:

evaluation = evaluation1 + evaluation2

#Return it:

return evaluation

def pieceSquareTable(self,flatboard,gamephase):

#Initialize score:

p=PieceTable()

score = 0

#Go through each square:

for i in range(64):

if flatboard[i]==0:

#Empty square

continue

#Get data:

piece = flatboard[i][0]

color = flatboard[i][1]

sign = +1

#Adjust index if black piece, since piece sqaure tables

#were designed for white:

if color=='b':

i = ((7-i)//8)\*8 + i%8

sign = -1

#Adjust score:

if piece=='P':

score += sign\*p.pawn\_table[i]

elif piece=='N':

score+= sign\*p.knight\_table[i]

elif piece=='B':

score+=sign\*p.bishop\_table[i]

elif piece=='R':

score+=sign\*p.rook\_table[i]

elif piece=='Q':

score+=sign\*p.queen\_table[i]

elif piece=='K':

#King has different table values based on phase

#of the game:

if gamephase=='opening':

score+=sign\*p.king\_table[i]

else:

score+=sign\*p.king\_endgame\_table[i]

return score

def doubledPawns(self,board,color):

color = color[0]

#Get indices of pawns:

listofpawns = self.c.lookfor(board,'P'+color)

#Count the number of doubled pawns by counting occurences of

#repeats in their x-coordinates:

repeats = 0

temp = []

for pawnpos in listofpawns:

if pawnpos[0] in temp:

repeats = repeats + 1

else:

temp.append(pawnpos[0])

return repeats

def blockedPawns(self,board,color):

color = color[0]

listofpawns = self.c.lookfor(board,'P'+color)

blocked = 0

#Self explanatory:

for pawnpos in listofpawns:

if ((color=='w' and self.c.isOccupiedby(board,pawnpos[0],pawnpos[1]-1,

'black'))

or (color=='b' and self.c.isOccupiedby(board,pawnpos[0],pawnpos[1]+1,

'white'))):

blocked = blocked + 1

return blocked

def isolatedPawns(self,board,color):

color = color[0]

listofpawns = self.c.lookfor(board,'P'+color)

#Get x coordinates of all the pawns:

xlist = [x for (x,y) in listofpawns]

isolated = 0

for x in xlist:

if x!=0 and x!=7:

#For non-edge cases:

if x-1 not in xlist and x+1 not in xlist:

isolated+=1

elif x==0 and 1 not in xlist:

#Left edge:

isolated+=1

elif x==7 and 6 not in xlist:

#Right edge:

isolated+=1

return isolated

#Initialize the board:

class Board:

def \_\_init\_\_(self):

self.create\_board()

def create\_board(self):

self.chess=[[0]\*8 for i in range(8)]

list\_w=['Rw','Nw','Bw','Qw','Kw','Bw','Nw','Rw']

list\_b=['Rb','Nb','Bb','Qb','Kb','Bb','Nb','Rb']

for i in range(2):

for j in range (8):

if i==0:

self.chess[i][j]=list\_b[j]

else:

self.chess[i][j]='Pb'

for i in range(6,8):

for j in range (8):

if i==7:

self.chess[i][j]=list\_w[j]

else:

self.chess[i][j]='Pw'

def getChess(self):

return self.chess

class PieceTable:

def \_\_init\_\_(self):

self.pawn\_table = [ 0, 0, 0, 0, 0, 0, 0, 0,

50, 50, 50, 50, 50, 50, 50, 50,

10, 10, 20, 30, 30, 20, 10, 10,

5, 5, 10, 25, 25, 10, 5, 5,

0, 0, 0, 20, 20, 0, 0, 0,

5, -5,-10, 0, 0,-10, -5, 5,

5, 10, 10,-20,-20, 10, 10, 5,

0, 0, 0, 0, 0, 0, 0, 0]

self.knight\_table = [-50,-40,-30,-30,-30,-30,-40,-50,

-40,-20, 0, 0, 0, 0,-20,-40,

-30, 0, 10, 15, 15, 10, 0,-30,

-30, 5, 15, 20, 20, 15, 5,-30,

-30, 0, 15, 20, 20, 15, 0,-30,

-30, 5, 10, 15, 15, 10, 5,-30,

-40,-20, 0, 5, 5, 0,-20,-40,

-50,-90,-30,-30,-30,-30,-90,-50]

self.bishop\_table = [-20,-10,-10,-10,-10,-10,-10,-20,

-10, 0, 0, 0, 0, 0, 0,-10,

-10, 0, 5, 10, 10, 5, 0,-10,

-10, 5, 5, 10, 10, 5, 5,-10,

-10, 0, 10, 10, 10, 10, 0,-10,

-10, 10, 10, 10, 10, 10, 10,-10,

-10, 5, 0, 0, 0, 0, 5,-10,

-20,-10,-90,-10,-10,-90,-10,-20]

self.rook\_table = [0, 0, 0, 0, 0, 0, 0, 0,

5, 10, 10, 10, 10, 10, 10, 5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

-5, 0, 0, 0, 0, 0, 0, -5,

0, 0, 0, 5, 5, 0, 0, 0]

self.queen\_table = [-20,-10,-10, -5, -5,-10,-10,-20,

-10, 0, 0, 0, 0, 0, 0,-10,

-10, 0, 5, 5, 5, 5, 0,-10,

-5, 0, 5, 5, 5, 5, 0, -5,

0, 0, 5, 5, 5, 5, 0, -5,

-10, 5, 5, 5, 5, 5, 0,-10,

-10, 0, 5, 0, 0, 0, 0,-10,

-20,-10,-10, 70, -5,-10,-10,-20]

self.king\_table = [-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-30,-40,-40,-50,-50,-40,-40,-30,

-20,-30,-30,-40,-40,-30,-30,-20,

-10,-20,-20,-20,-20,-20,-20,-10,

20, 20, 0, 0, 0, 0, 20, 20,

20, 30, 10, 0, 0, 10, 30, 20]

self.king\_endgame\_table = [-50,-40,-30,-20,-20,-30,-40,-50,

-30,-20,-10, 0, 0,-10,-20,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 30, 40, 40, 30,-10,-30,

-30,-10, 20, 30, 30, 20,-10,-30,

-30,-30, 0, 0, 0, 0,-30,-30,

-50,-30,-30,-30,-30,-30,-30,-50]

##############################////////GUI FUNCTIONS\\\\\\\\\\\\\#############################

#########MAIN FUNCTION####################################################

class GUI:

def \_\_init\_\_(self):

self.board = Board().getChess()

self.c = Commands()

self.a = AI()

#In chess some data must be stored that is not apparent in the board:

self.player = 0 #This is the player that makes the next move. 0 is white, 1 is black

self.castling\_rights = [[True, True],[True, True]]

#The above stores whether or not each of the players are permitted to castle on

#either side of the king. (Kingside, Queenside)

self.En\_Passant\_Target = -1 #This variable will store a coordinate if there is a square that can be

#en passant captured on. Otherwise it stores -1, indicating lack of en passant

#targets.

self.half\_move\_clock = 0 #This variable stores the number of reversible moves that have been played so far.

#Generate an instance of GamePosition class to store the above data:

self.position = GamePosition(self.board,self.player,self.castling\_rights,self.En\_Passant\_Target

,self.half\_move\_clock)

#Store the piece square tables here so they can be accessed globally by pieceSquareTable() function:

pygame.init()

self.size = (640, 640)

self.screen = pygame.display.set\_mode(self.size)

pygame.display.set\_caption("Chess Game Press \*e\* to Exit")

self.game\_icon = pygame.image.load('newMedia/ChessImage.png')

pygame.display.set\_icon(self.game\_icon)

self.media()

self.bg = (49, 60, 43)

self.startPage = pygame.Surface(self.size)

self.startPage.fill(self.bg)

self.diffPage = pygame.Surface(self.size)

self.diffPage.fill(self.bg)

self.flipPage = pygame.Surface(self.size)

self.flipPage.fill(self.bg)

self.selectPage = pygame.Surface(self.size)

self.selectPage.fill(self.bg)

self.colorPage = pygame.Surface(self.size)

self.colorPage.fill(self.bg)

# Stored [ x , y , width , height ] of buttons

self.buttons = {

1: [460-275, 380-15, 280, 75],

2: [460-275, 470-15, 280, 75],

3: [325-275, 280-15, 250, 250],

4: [625-275, 280-15, 250, 250],

5: [309-275, 250-15, 180, 180],

6: [509-275, 250-15, 180, 180],

7: [709-275, 250-15, 180, 180],

8: [460-275, 560-15, 280, 75]

}

self.diffMenu = -1

self.select = -1

self.level = None

self.temp = None

self.box = pygame.image.load('newMedia/box.png')

self.box = pygame.transform.scale(self.box, (640, 640))

self.screen.blit(self.box,(0,0))

pygame.mixer.Sound.play(self.welcome\_sound)

clock = pygame.time.Clock() # Helps controlling fps of the game.

self.initialize()

pygame.display.update()

#########################INFINITE LOOP#####################################

#The program remains in this loop until the user quits the application

while not self.gameEnded:

if self.isMenu:

#Menu needs to be shown right now.

#Blit the background:

#self.screen.blit(self.background,(0,0))

if self.isAI==-1:

self.startMenu()

elif self.isAI==True:

if self.diffMenu == -1:

self.play1Menu\_A()

elif self.diffMenu == 1:

self.play1Menu\_B()

if self.select == 1 and self.temp == None:

self.selectMenu()

elif self.isAI==False:

self.selectMenu()

if self.isFlip!=-1 and self.select == 2 :

self.call\_board()

continue

elif self.isFlip!=-1 and self.select == 3 :

self.call\_board()

pygame.mixer.Sound.play(self.instructions\_sound)

continue

if self.isFlip!=-1 and self.temp == -1 :

self.call\_board()

continue

for event in pygame.event.get():

#Handle the events while in menu:

if event.type == KEYDOWN:

if event.key == K\_e :

self.gameEnded = True

elif event.type == QUIT:

self.gameEnded = True

#Window was closed.

#self.gameEnded = True

pygame.mixer.Sound.play(self.exit\_sound)

break

if event.type == MOUSEBUTTONUP:

self.onClick()

#Update the display:

pygame.display.update()

#Run at specific fps:

clock.tick(10)

continue

#Menu part was done if this part reached.

#If the AI is currently thinking the move to play

#next, show some fancy looking squares to indicate

#that.

#Do it every 6 frames so it's not too fast:

self.numm+=1

if self.isAIThink and self.numm%10==0:

self.Thinking()

for event in pygame.event.get():

#Deal with all the user inputs:

if event.type == KEYDOWN:

if event.key == K\_e:

self.gameEnded = True

#Window was closed.

elif event.type == QUIT:

self.gameEnded = True

#self.gameEnded = True

pygame.mixer.Sound.play(self.exit\_sound)

break

#Under the following conditions, user input should be

#completely ignored:

if self.chessEnded or self.isTransition or self.isAIThink:

continue

#isDown means a piece is being dragged.

if self.select<=2:

if not self.isDown and event.type == MOUSEBUTTONDOWN:

#Mouse was pressed down.

#Get the oordinates of the mouse

pos = pygame.mouse.get\_pos()

if pos[0] in range(0,640) and pos[1] in range(0,640):

#convert to chess coordinates:

chess\_coord = self.pixel\_coord\_to\_chess(pos)

x = chess\_coord[0]

y = chess\_coord[1]

#If the piece clicked on is not occupied by your own piece,

#ignore this mouse click:

if not self.c.isOccupiedby(self.board,x,y,'wb'[self.player]):

continue

#Now we're sure the user is holding their mouse on a

#piecec that is theirs.

#Get reference to the piece that should be dragged around or selected:

dragPiece = self.getPiece(chess\_coord)

#Find the possible squares that this piece could attack:

listofTuples = self.c.findPossibleSquares(self.position,x,y)

#Highlight all such squares:

self.createShades(listofTuples)

#A green box should appear on the square which was selected, unless

#it's a king under check, in which case it shouldn't because the king

#has a red color on it in that case.

if dragPiece:

if ((dragPiece.pieceinfo[0]=='K') and

(self.c.isCheck(self.position,'white') or self.c.isCheck(self.position,'black'))):

None

else:

self.listofShades.append(Shades(self.greenbox\_image,(x,y)))

#A piece is being dragged:

self.isDown = True

if (self.isDown or self.isClicked) and event.type == MOUSEBUTTONUP:

#Mouse was released.

self.isDown = False

#Snap the piece back to its coordinate position

if dragPiece:

dragPiece.setpos((-1,-1))

#Get coordinates and convert them:

pos = pygame.mouse.get\_pos()

chess\_coord = self.pixel\_coord\_to\_chess(pos)

x2 = chess\_coord[0]

y2 = chess\_coord[1]

#Initialize:

self.isTransition = False

if (x,y)==(x2,y2): #NO dragging occured

#(ie the mouse was held and released on the same square)

if not self.isClicked: #nothing had been clicked previously

#This is the first click

self.isClicked = True

self.prevPos = (x,y) #Store it so next time we know the origin

else: #Something had been clicked previously

#Find out location of previous click:

x,y = self.prevPos

if (x,y)==(x2,y2): #User clicked on the same square again.

#So

self.isClicked = False

#Destroy all shades:

window = Tk()

window.wm\_withdraw()

window.geometry("1x1+200+200")

tkinter.messagebox.showinfo(title="Invalid move",message="Invalid move",parent=window)

else:

#User clicked elsewhere on this second click:

if self.c.isOccupiedby(self.board,x2,y2,'wb'[self.player]):

#User clicked on a square that is occupied by their

#own piece.

#This is like making a first click on your own piece:

self.isClicked = True

self.prevPos = (x2,y2) #Store it

else:

#The user may or may not have clicked on a valid target square.

self.isClicked = False

#Destory all shades

if not (x2,y2) in listofTuples:

window = Tk()

window.wm\_withdraw()

window.geometry("1x1+200+200")

tkinter.messagebox.showinfo(title="Invalid move",message="Invalid move",parent=window)

if not (x2,y2) in listofTuples:

#Move was invalid

self.isTransition = False

continue

#Reaching here means a valid move was selected.

#If the recording option was selected, store the move to the opening dictionary:

if self.isRecord:

key = self.c.pos2key(self.position)

#Make sure it isn't already in there:

if [(x,y),(x2,y2)] not in self.openings[key]:

self.openings[key].append([(x,y),(x2,y2)])

#Make the move:

self.c.makemove(self.position,x,y,x2,y2)

#Update this move to be the 'previous' move (latest move in fact), so that

#yellow shades can be shown on it.

self.prevMove = [x,y,x2,y2]

#Update which player is next to play:

self.player = self.position.getplayer()

if self.player == 1:

pygame.mixer.Sound.play(self.piece\_sound)

else:

pygame.mixer.Sound.play(self.piece\_sound)

#Add the new position to the history for it:

self.position.addtoHistory(self.position)

#Check for possibilty of draw:

HMC = self.position.getHMC()

if HMC>=100 or self.c.isStalemate(self.position) or self.position.checkRepition():

#There is a draw:

self.isDraw = True

self.chessEnded = True

#Check for possibilty of checkmate:

if self.c.isCheckmate(self.position,'white'):

self.winner = 'b'

self.chessEnded = True

if self.c.isCheckmate(self.position,'black'):

self.winner = 'w'

self.chessEnded = True

#If the AI option was selected and the game still hasn't finished,

#let the AI start thinking about its next move:

if self.isAI and not self.chessEnded:

if self.player==0:

colorsign = 1

else:

colorsign = -1

self.bestMoveReturn = []

self.move\_thread = threading.Thread(target = self.a.negamax,

args = (self.position,self.level,-1000000,1000000,colorsign,self.bestMoveReturn,self.openings,self.searched))

self.move\_thread.start()

self.isAIThink = True

#Move the piece to its new destination:

dragPiece.setcoord((x2,y2))

#There may have been a capture, so the piece list should be regenerated.

#However, if animation is ocurring, the the captured piece should still remain visible.

if not self.isTransition:

self.listofWhitePieces,self.listofBlackPieces = self.createPieces(self.board)

else:

movingPiece = dragPiece

origin = self.chess\_coord\_to\_pixels((x,y))

print("Source :",(y,x))

destiny = self.chess\_coord\_to\_pixels((x2,y2))

print("Destination:",(y2,x2))

print("\n")

movingPiece.setpos(origin)

step = (destiny[0]-origin[0],destiny[1]-origin[1])

#Either way shades should be deleted now:

self.createShades([])

else:

if event.type == pygame.MOUSEBUTTONDOWN and event.button==1:

if self.player==1:

self.letters\_dict = {'a': 7, 'b': 6, 'c': 5, 'd': 4, 'e': 3, 'f': 2, 'g': 1, 'h': 0}

self.numbers\_dict = {'1': 0, '2': 1, '3': 2, '4': 3, '5': 4, '6': 5, '7': 6, '8': 7}

with sr.Microphone() as source:

self.r.adjust\_for\_ambient\_noise(source)

pygame.mixer.Sound.play(self.selectpiece\_sound)

time.sleep(1.5)

try:

audio = self.r.listen(source,timeout=2,phrase\_time\_limit=2)

print("Recognizing...")

query = self.r.recognize\_google(audio)

print(f"User said: {query}\n")

voice = query.lower()

if voice=='avon' or voice=='ye one' or voice=='evan' or voice=='yah 1':

voice='a1'

elif voice == 'heetu' or voice=='hetu' or voice=='do' or voice =='tattoo' or voice =='airport' or voice =='tetu' or voice =='edu':

voice='a2'

elif voice == 'a tree' or voice=='83' or voice=='yatri':

voice='a3'

elif voice == 'krrish 4':

voice='a4'

elif voice=='before':

voice='b4'

elif voice=='bittu' or voice=='titu':

voice='b2'

elif voice=='ba' or voice=='b.ed':

voice='b8'

elif voice=='shivan' or voice=='shiva' or voice=='civil':

voice='c1'

elif voice=='ceat':

voice='c8'

elif voice=='deewan' or voice=='d 1' or voice=='devon' or voice=='devil':

voice='d1'

elif voice=='even' or voice=='evil' or voice=='evan' or voice=='yuvan' or voice=='t1':

voice='e1'

elif voice=='youtube' or voice=='tu':

voice='e2'

elif voice=='mi 4':

voice='e4'

elif voice=='mi 5':

voice='e5'

elif voice=='8':

voice='e8'

elif voice=='jivan':

voice='g1'

elif voice=='jeetu' or voice=='jitu':

voice='g2'

elif voice=='zefo':

voice='g4'

elif voice=='quit' or voice =='end' or voice == 'close' or voice=='stop' or voice=='friend' or voice=='top' or voice=='finish' or voice=='and':

pygame.mixer.Sound.play(self.exit\_sound)

self.gameEnded=True

if len(voice) == 2:

letter = voice[0]

number = voice[1]

if letter=='v':

letter='b'

elif letter=='s':

letter='h'

if letter in self.letters\_dict.keys() and number in self.numbers\_dict.keys():

print(self.letters\_dict[letter], self.numbers\_dict[number])

chess\_coord = (self.letters\_dict[letter], self.numbers\_dict[number])

x = chess\_coord[0]

y = chess\_coord[1]

# If the piece clicked on is not occupied by your own piece,

# ignore this mouse click:

if not self.c.isOccupiedby(self.board, x, y, 'wb'[self.player]):

continue

# Now we're sure the user is holding their mouse on a

# piecec that is theirs.

# Get reference to the piece that should be dragged around or selected:

dragPiece = self.getPiece(chess\_coord)

# Find the possible squares that this piece could attack:

listofTuples = self.c.findPossibleSquares(self.position, x, y)

# Highlight all such squares:

self.createShades(listofTuples)

# A green box should appear on the square which was selected, unless

# it's a king under check, in which case it shouldn't because the king

# has a red color on it in that case.

if dragPiece:

if ((dragPiece.pieceinfo[0] == 'K') and

(self.c.isCheck(self.position, 'white') or self.c.isCheck(self.position,

'black'))):

None

else:

self.listofShades.append(Shades(self.greenbox\_image, (x, y)))

self.piece\_selected\_by\_voice = True

except sr.UnknownValueError:

pygame.mixer.Sound.play(self.repeat\_sound)

except sr.RequestError:

pygame.mixer.Sound.play(self.requesterror\_sound)

except Exception:

pygame.mixer.Sound.play(self.repeat\_sound)

#Move to Destination Using voice

elif self.piece\_selected\_by\_voice and event.type==pygame.MOUSEBUTTONDOWN and event.button==3 :

self.piece\_selected\_by\_voice = False

with sr.Microphone() as source:

while True:

self.r.adjust\_for\_ambient\_noise(source)

pygame.mixer.Sound.play(self.destination\_sound)

time.sleep(1.5)

try:

audio = self.r.listen(source,timeout=2,phrase\_time\_limit=2)

print("Recognizing...")

query2 = self.r.recognize\_google(audio)

print(f"User said: {query2}\n")

voice2 = query2.lower()

if voice2=='avon':

voice2='a1'

elif voice2 == 'heetu' or voice2=='hetu' or voice2=='do' or voice2 =='tattoo' or voice2 =='airport' or voice2 =='tetu' or voice2 =='edu':

voice2='a2'

elif voice2 == 'a tree' or voice2=='83':

voice2='a3'

elif voice2 == 'krrish 4':

voice2='a4'

elif voice2=='before':

voice2='b4'

elif voice2=='bittu' or voice2=='titu':

voice2='b2'

elif voice2=='ba' or voice2=='b.ed':

voice2='b8'

elif voice2=='shivan' or voice2=='shiva' or voice2=='civil':

voice2='c1'

elif voice2=='ceat':

voice2='c8'

elif voice2=='deewan' or voice2=='d 1' or voice2=='devon' or voice2=='devil':

voice2='d1'

elif voice2=='even' or voice2=='evil' or voice2=='evan' or voice2=='yuvan' or voice2=='t1':

voice='e1'

elif voice2=='youtube' or voice2=='tu':

voice2='e2'

elif voice2=='mi 4':

voice2='e4'

elif voice2=='mi 5':

voice2='e5'

elif voice2=='8':

voice2='e8'

elif voice2=='jivan':

voice2='g1'

elif voice2=='jeetu' or voice2=='jitu':

voice2='g2'

elif voice2=='zefo':

voice2='g4'

elif voice2 == 'quit' or voice2 == 'end' or voice2 == 'close' or voice2=='stop' or voice2=='friend' or voice2=='top' or voice2=='finish' or voice2=='and':

pygame.mixer.Sound.play(self.exit\_sound)

self.gameEnded=True

break

if len(voice2) == 2:

letter = voice2[0]

number = voice2[1]

if letter=='v':

letter='b'

elif letter=='s':

letter='h'

if letter in self.letters\_dict.keys() and number in self.numbers\_dict.keys():

print(self.letters\_dict[letter], self.numbers\_dict[number])

chess\_coord = (self.letters\_dict[letter], self.numbers\_dict[number])

x2 = chess\_coord[0]

y2 = chess\_coord[1]

# Initialize:

self.isTransition = False

if not (x2, y2) in listofTuples:

# Move was invalid

self.isTransition = False

pygame.mixer.Sound.play(self.invalid\_sound)

continue

# Reaching here means a valid move was selected.

# If the recording option was selected, store the move to the opening dictionary:

if self.isRecord:

key = self.c.pos2key(self.position)

# Make sure it isn't already in there:

if [(x, y), (x2, y2)] not in self.openings[key]:

self.openings[key].append([(x, y), (x2, y2)])

# Make the move:

self.c.makemove(self.position, x, y, x2, y2)

# Update this move to be the 'previous' move (latest move in fact), so that

# yellow shades can be shown on it.

self.prevMove = [x, y, x2, y2]

# Update which player is next to play:

self.player = self.position.getplayer()

if self.player == 1:

pygame.mixer.Sound.play(self.piece\_sound)

else:

pygame.mixer.Sound.play(self.piece\_sound)

# Add the new position to the history for it:

self.position.addtoHistory(self.position)

# Check for possibilty of draw:

HMC = self.position.getHMC()

if HMC >= 100 or self.c.isStalemate(self.position) or self.position.checkRepition():

# There is a draw:

self.isDraw = True

self.chessEnded = True

# Check for possibilty of checkmate:

if self.c.isCheckmate(self.position, 'white'):

self.winner = 'b'

self.chessEnded = True

if self.c.isCheckmate(self.position, 'black'):

self.winner = 'w'

self.chessEnded = True

# If the AI option was selected and the game still hasn't finished,

# let the AI start thinking about its next move:

if self.isAI and not self.chessEnded:

if self.player == 0:

colorsign = 1

else:

colorsign = -1

self.bestMoveReturn = []

self.move\_thread = threading.Thread(target=self.a.negamax,

args=(self.position, self.level, -1000000, 1000000,

colorsign,

self.bestMoveReturn, self.openings,

self.searched))

self.move\_thread.start()

self.isAIThink = True

# Move the piece to its new destination:

dragPiece.setcoord((x2, y2))

# There may have been a capture, so the piece list should be regenerated.

# However, if animation is ocurring, the the captured piece should still remain visible.

if not self.isTransition:

self.listofWhitePieces, self.listofBlackPieces = self.createPieces(self.board)

else:

movingPiece = dragPiece

origin = self.chess\_coord\_to\_pixels((x, y))

destiny = self.chess\_coord\_to\_pixels((x2, y2))

movingPiece.setpos(origin)

step = (destiny[0] - origin[0], destiny[1] - origin[1])

# Either way shades should be deleted now:

self.createShades([])

break

except sr.UnknownValueError:

pygame.mixer.Sound.play(self.repeat\_sound)

except sr.RequestError:

pygame.mixer.Sound.play(self.requesterror\_sound)

except Exception:

pygame.mixer.Sound.play(self.repeat\_sound)

#If an animation is supposed to happen, make it happen:

if self.isTransition:

p,q = movingPiece.getpos()

dx2,dy2 = destiny

n= 30.0

if abs(p-dx2)<=abs(step[0]/n) and abs(q-dy2)<=abs(step[1]/n):

#The moving piece has reached its destination:

#Snap it back to its grid position:

movingPiece.setpos((-1,-1))

#Generate new piece list in case one got captured:

self.listofWhitePieces,self.listofBlackPieces = self.createPieces(self.board)

#No more transitioning:

self.isTransition = False

self.createShades([])

else:

#Move it closer to its destination.

movingPiece.setpos((p+step[0]/n,q+step[1]/n))

#If a piece is being dragged let the dragging piece follow the mouse:

if self.isDown:

m,k = pygame.mouse.get\_pos()

if dragPiece:

dragPiece.setpos((m-self.square\_width/2,k-self.square\_height/2))

#If the AI is thinking, make sure to check if it isn't done thinking yet.

#Also, if a piece is currently being animated don't ask the AI if it's

#done thining, in case it replied in the affirmative and starts moving

#at the same time as your piece is moving:

if self.isAIThink and not self.isTransition:

if not self.move\_thread.is\_alive():

#The AI has made a decision.

#It's no longer thinking

self.isAIThink = False

#Destroy any shades:

self.createShades([])

#Get the move proposed:

if len(self.bestMoveReturn)==2:

[x,y],[x2,y2] = self.bestMoveReturn

else:

self.c.allMoves(self.position,color)

#Do everything just as if the user made a move by click-click movement:

self.c.makemove(self.position,x,y,x2,y2)

self.prevMove = [x,y,x2,y2]

self.player = self.position.getplayer()

HMC = self.position.getHMC()

self.position.addtoHistory(self.position)

if HMC>=100 or self.c.isStalemate(self.position) or self.position.checkRepition():

self.isDraw = True

self.chessEnded = True

if self.c.isCheckmate(self.position,'white'):

self.winner = 'b'

self.chessEnded = True

if self.c.isCheckmate(self.position,'black'):

self.winner = 'w'

self.chessEnded = True

#Animate the movement:

self.isTransition = True

movingPiece = self.getPiece((x,y))

origin = self.chess\_coord\_to\_pixels((x,y))

destiny = self.chess\_coord\_to\_pixels((x2,y2))

movingPiece.setpos(origin)

step = (destiny[0]-origin[0],destiny[1]-origin[1])

pygame.mixer.Sound.play(self.piece\_sound)

#Update positions of all images:

self.drawBoard()

#Update the display:

pygame.display.update()

#Run at specific fps:

clock.tick(60)

#Out of loop. Quit pygame:

time.sleep(2)

pygame.quit()

#In case recording mode was on, save the openings dictionary to a file:

if self.isRecord:

file\_handle.seek(0)

pickle.dump(self.openings,file\_handle)

file\_handle.truncate()

file\_handle.close()

def DisplayPage(self, pageName):

self.SurfacesAtTop = self.SurfacesAtTop.fromkeys(self.SurfacesAtTop, False)

self.screen.blit(self.Surfaces[pageName], (0, 0))

self.SurfacesAtTop[pageName] = True

def chess\_coord\_to\_pixels(self,chess\_coord):

x,y = chess\_coord

#There are two sets of coordinates that this function could choose to return.

#One is the coordinates that would be usually returned, the other is one that

#would be returned if the board were to be flipped.

#Note that square width and height variables are defined in the main function and

#so are accessible here as global variables.

if self.isAI:

if self.AIPlayer==0:

#This means you're playing against the AI and are playing as black:

return ((7-x)\*self.square\_width, (7-y)\*self.square\_height)

else:

return (x\*self.square\_width, y\*self.square\_height)

#Being here means two player game is being played.

#If the flipping mode is enabled, and the player to play is black,

#the board should flip, but not until the transition animation for

#white movement is complete:

if not self.isFlip or self.player==0 ^ self.isTransition:

return (x\*self.square\_width, y\*self.square\_height)

else:

return ((7-x)\*self.square\_width, (7-y)\*self.square\_height)

def pixel\_coord\_to\_chess(self,pixel\_coord):

if pixel\_coord[0] in range(0,640) and pixel\_coord[1] in range(0,640):

x,y = (pixel\_coord[0])//self.square\_width, (pixel\_coord[1])//self.square\_height

#See comments for chess\_coord\_to\_pixels() for an explanation of the

#conditions seen here:

if self.isAI:

if self.AIPlayer==0:

return (7-x,7-y)

else:

return (x,y)

if not self.isFlip or self.player==0 ^ self.isTransition:

return (x,y)

else:

return (7-x,7-y)

def getPiece(self,chess\_coord):

for piece in self.listofWhitePieces+self.listofBlackPieces:

#piece.getInfo()[0] represents the chess coordinate occupied

#by piece.

if piece.getInfo()[0] == chess\_coord:

return piece

def createPieces(self,board):

#Initialize containers:

self.listofWhitePieces = []

self.listofBlackPieces = []

#Loop through all squares:

for i in range(8):

for k in range(8):

if board[i][k]!=0:

#The square is not empty, create a piece object:

p = Piece(board[i][k],(k,i), self.square\_width, self.square\_height)

#Append the reference to the object to the appropriate

#list:

if board[i][k][1]=='w':

self.listofWhitePieces.append(p)

else:

self.listofBlackPieces.append(p)

#Return both:

return [self.listofWhitePieces,self.listofBlackPieces]

def createShades(self,listofTuples):

#Empty the list

self.listofShades = []

if self.isTransition:

#Nothing should be shaded when a piece is being animated:

return

if self.isDraw:

#The game ended with a draw. Make yellow circle shades for

#both the kings to show this is the case:

coord = self.c.lookfor(self.board,'Kw')[0]

shade = Shades(self.circle\_image\_yellow,coord)

self.listofShades.append(shade)

coord = self.c.lookfor(self.board,'Kb')[0]

shade = Shades(self.circle\_image\_yellow,coord)

self.listofShades.append(shade)

pygame.mixer.Sound.play(self.draw\_sound)

#There is no need to go further:

return

if self.chessEnded:

#The game has ended, with a checkmate because it cannot be a

#draw if the code reached here.

#Give the winning king a green circle shade:

coord = self.c.lookfor(self.board,'K'+self.winner)[0]

shade = Shades(self.circle\_image\_green\_big,coord)

self.listofShades.append(shade)

if self.winner=='w':

pygame.mixer.Sound.play(self.whitewin\_sound)

else:

pygame.mixer.Sound.play(self.blackwin\_sound)

#If either king is under attack, give them a red circle:

if self.c.isCheck(self.position,'white'):

coord = self.c.lookfor(self.board,'Kw')[0]

shade = Shades(self.circle\_image\_red,coord)

self.listofShades.append(shade)

pygame.mixer.Sound.play(self.checkmate\_sound)

if self.c.isCheck(self.position,'black'):

coord = self.c.lookfor(self.board,'Kb')[0]

shade = Shades(self.circle\_image\_red,coord)

self.listofShades.append(shade)

pygame.mixer.Sound.play(self.checkmate\_sound)

#Go through all the target squares inputted:

for pos in listofTuples:

#If the target square is occupied, it can be captured.

#For a capturable square, there is a different shade.

#Create the appropriate shade for each target square:

if self.c.isOccupied(self.board,pos[0],pos[1]):

img = self.circle\_image\_capture

else:

img = self.circle\_image\_green

shade = Shades(img,pos)

#Append:

self.listofShades.append(shade)

def drawBoard(self):

#Blit the background:

self.screen.blit(self.background,(0,0))

#Choose the order in which to blit the pieces.

#If black is about to play for example, white pieces

#should be blitted first, so that when black is capturing,

#the piece appears above:

if self.player==1:

order = [self.listofWhitePieces,self.listofBlackPieces]

else:

order = [self.listofBlackPieces,self.listofWhitePieces]

if self.isTransition:

#If a piece is being animated, the player info is changed despite

#white still capturing over black, for example. Reverse the order:

order = list(reversed(order))

#The shades which appear during the following three conditions need to be

#blitted first to appear under the pieces:

if self.isDraw or self.chessEnded or self.isAIThink:

#Shades

for shade in self.listofShades:

img,chess\_coord = shade.getInfo()

pixel\_coord = self.chess\_coord\_to\_pixels(chess\_coord)

self.screen.blit(img,pixel\_coord)

#Make shades to show what the previous move played was:

if self.prevMove[0]!=-1 and not self.isTransition:

x,y,x2,y2 = self.prevMove

self.screen.blit(self.yellowbox\_image,self.chess\_coord\_to\_pixels((x,y)))

self.screen.blit(self.yellowbox\_image,self.chess\_coord\_to\_pixels((x2,y2)))

#Blit the Pieces:

#Notw that one side has to be below the green circular shades to show

#that they are being targeted, and the other side if dragged to such

# a square should be blitted on top to show that it is capturing:

#Potentially captured pieces:

for piece in order[0]:

chess\_coord,subsection,pos = piece.getInfo()

pixel\_coord = self.chess\_coord\_to\_pixels(chess\_coord)

if pos==(-1,-1):

#Blit to default square:

self.screen.blit(self.pieces\_image,pixel\_coord,subsection)

else:

#Blit to the specific coordinates:

self.screen.blit(self.pieces\_image,pos,subsection)

#Blit the shades in between:

if not (self.isDraw or self.chessEnded or self.isAIThink):

for shade in self.listofShades:

img,chess\_coord = shade.getInfo()

pixel\_coord = self.chess\_coord\_to\_pixels(chess\_coord)

self.screen.blit(img,pixel\_coord)

#Potentially capturing pieces:

for piece in order[1]:

chess\_coord,subsection,pos = piece.getInfo()

pixel\_coord = self.chess\_coord\_to\_pixels(chess\_coord)

if pos==(-1,-1):

#Default square

self.screen.blit(self.pieces\_image,pixel\_coord,subsection)

else:

#Specifc pixels:

self.screen.blit(self.pieces\_image,pos,subsection)

def media(self):

#Load all the images:

#Load the background chess board image:

self.background = pygame.image.load('Media\\board2.png').convert()

#Load an image with all the pieces on it:

pieces\_image = pygame.image.load('Media\\Chess\_Pieces\_Sprite.png').convert\_alpha()

circle\_image\_green = pygame.image.load('Media\\green\_circle\_small.png').convert\_alpha()

circle\_image\_capture = pygame.image.load('Media\\green\_circle\_neg.png').convert\_alpha()

circle\_image\_red = pygame.image.load('Media\\red\_circle\_big.png').convert\_alpha()

greenbox\_image = pygame.image.load('Media\\green\_box.png').convert\_alpha()

circle\_image\_yellow = pygame.image.load('Media\\yellow\_circle\_big.png').convert\_alpha()

circle\_image\_green\_big = pygame.image.load('Media\\green\_circle\_big.png').convert\_alpha()

yellowbox\_image = pygame.image.load('Media\\yellow\_box.png').convert\_alpha()

#Getting sizes:

#Get background size:

self.size\_of\_bg = self.background.get\_rect().size

#Get size of the individual squares

self.square\_width = self.size\_of\_bg[0]//8

self.square\_height = self.size\_of\_bg[1]//8

#Rescale the images so that each piece can fit in a square:

self.pieces\_image = pygame.transform.scale(pieces\_image,

(self.square\_width\*6,self.square\_height\*2))

self.circle\_image\_green = pygame.transform.scale(circle\_image\_green,

(self.square\_width, self.square\_height))

self.circle\_image\_capture = pygame.transform.scale(circle\_image\_capture,

(self.square\_width, self.square\_height))

self.circle\_image\_red = pygame.transform.scale(circle\_image\_red,

(self.square\_width, self.square\_height))

self.greenbox\_image = pygame.transform.scale(greenbox\_image,

(self.square\_width, self.square\_height))

self.yellowbox\_image = pygame.transform.scale(yellowbox\_image,

(self.square\_width, self.square\_height))

self.circle\_image\_yellow = pygame.transform.scale(circle\_image\_yellow,

(self.square\_width, self.square\_height))

self.circle\_image\_green\_big = pygame.transform.scale(circle\_image\_green\_big,

(self.square\_width, self.square\_height))

#Loading Sounds

self.welcome\_sound = pygame.mixer.Sound("Voice\welcome.wav")

self.exit\_sound = pygame.mixer.Sound("Voice\exit.wav")

self.flip\_sound = pygame.mixer.Sound("Voice\Flip.wav")

self.color\_sound = pygame.mixer.Sound("Voice\color.wav")

self.thinking\_sound = pygame.mixer.Sound("Voice\Thinking.wav")

self.difficulty\_sound=pygame.mixer.Sound("Voice\difficulty.wav")

self.turn\_sound=pygame.mixer.Sound("Voice\Turn.wav")

self.checkmate\_sound = pygame.mixer.Sound("Voice\check.wav")

self.draw\_sound = pygame.mixer.Sound("Voice\draw.wav")

self.whitewin\_sound = pygame.mixer.Sound("Voice\whitewins.wav")

self.blackwin\_sound = pygame.mixer.Sound("Voice\Blackwins.wav")

self.blackturn\_sound = pygame.mixer.Sound("Voice\Blackturn.wav")

self.whiteturn\_sound = pygame.mixer.Sound("Voice\whiteturn.wav")

self.piece\_sound=pygame.mixer.Sound("Voice\piecehit.wav")

self.destination\_sound=pygame.mixer.Sound("Voice\destination.wav")

self.instructions\_sound = pygame.mixer.Sound("Voice\instructions.wav")

self.repeat\_sound = pygame.mixer.Sound("Voice\Repeat.wav")

self.selectpiece\_sound = pygame.mixer.Sound("Voice\selectpiece.wav")

self.requesterror\_sound = pygame.mixer.Sound("Voice\Requesterror.wav")

self.control\_sound = pygame.mixer.Sound("Voice\control.wav")

self.invalid\_sound = pygame.mixer.Sound("Voice\invalid.wav")

def initialize(self):

#Generate a list of pieces that should be drawn on the board:

self.listofWhitePieces,self.listofBlackPieces = self.createPieces(self.board)

#(the list contains references to objects of the class Piece)

#Initialize a list of shades:

self.listofShades = []

self.isDown = False #Variable that shows if the mouse is being held down

#onto a piece

self.isClicked = False #To keep track of whether a piece was clicked in order

#to indicate intention to move by the user.

self.isTransition = False #Keeps track of whether or not a piece is being animated.

self.isDraw = False #Will store True if the game ended with a draw

self.chessEnded = False #Will become True once the chess game ends by checkmate, stalemate, etc.

self.isRecord = False #Set this to True if you want to record moves to the Opening Book. Do not

#set this to True unless you're 100% sure of what you're doing. The program will never modify

#this value.

self.isAIThink = False #Stores whether or not the AI is calculating the best move to be played.

# Initialize the opening book dictionary, and set its values to be lists by default:

self.openings = defaultdict(list)

#If openingTable.txt exists, read from it and load the opening moves to the local dictionary.

#If it doesn't, create a new one to write to if Recording is enabled:

try:

file\_handle = open('openingTable.txt','r')

self.openings = pickle.loads(file\_handle.read())

except:

if self.isRecord:

file\_handle = open('openingTable.txt','w')

self.letters\_dict = {'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4, 'f': 5, 'g': 6, 'h': 7}#dictionary for voice

self.numbers\_dict = {'1': 7, '2': 6, '3': 5, '4': 4, '5': 3, '6': 2, '7': 1, '8': 0}#dictionary for voice

self.piece\_selected\_by\_voice=False

self.r = sr.Recognizer()#speechrecognition class object

self.r.dynamic\_energy\_threshold = False

self.r.energy\_threshold = 400

self.searched = {} #Global variable that allows negamax to keep track of nodes that have

#already been evaluated.

self.prevMove = [-1,-1,-1,-1] #Also a global varible that stores the last move played, to

#allow drawBoard() to create Shades on the squares.

#Initialize some more values:

#For animating AI thinking graphics:

self.ax,self.ay=0,0

self.numm = 0

#For showing the menu and keeping track of user choices:

self.isMenu = True

self.isAI = -1

self.isFlip = -1

self.AIPlayer = -1

#Finally, a variable to keep false until the user wants to quit:

self.gameEnded = False

def startMenu(self):

#The user has not selected between playing against the AI

#or playing against a friend.

#So allow them to choose between playing with a friend or the AI:

self.boardImage = pygame.image.load('newMedia/ChessImage.png')

self.boardImage = pygame.transform.scale(self.boardImage, (300, 300))

self.player1 = pygame.image.load('newMedia/play1.png')

self.player1 = pygame.transform.scale(self.player1, (280, 75))

self.player2 = pygame.image.load('newMedia/play2.png')

self.player2 = pygame.transform.scale(self.player2, (280, 75))

self.exit = pygame.image.load('newMedia/exit.png')

self.exit = pygame.transform.scale(self.exit, (280, 75))

self.startPage.blit(self.box,(0,0))

self.startPage.blit(self.boardImage, (450-275, 60-15))

self.startPage.blit(self.player1, (460-275, 380-15))

self.startPage.blit(self.player2, (460-275, 470-15))

self.startPage.blit(self.exit, (460-275, 560-15))

self.screen.blit(self.startPage, (0, 0))

def play1Menu\_A(self):

#The user has selected to play against the AI.

#Allow the user to play as white or black:

#self.screen.blit(self.playwhite\_pic,(0,self.square\_height\*2))

#self.screen.blit(self.playblack\_pic,(self.square\_width\*4,self.square\_height\*2))

self.selectcolor = pygame.image.load('newMedia/selectColor.png')

self.selectcolor = pygame.transform.scale(self.selectcolor, (350, 80))

self.playasblack = pygame.image.load('newMedia/playBlack.png')

self.playasblack = pygame.transform.scale(self.playasblack, (250, 250))

self.playaswhite = pygame.image.load('newMedia/playWhite.png')

self.playaswhite = pygame.transform.scale(self.playaswhite, (250, 250))

self.colorPage.blit(self.box,(0,0))

self.colorPage.blit(self.selectcolor, (425-275, 80-15))

self.colorPage.blit(self.playasblack, (325-275, 280-15))

self.colorPage.blit(self.playaswhite, (625-275, 280-15))

self.screen.blit(self.colorPage, (0, 0))

global play\_sound

if play\_sound:

play\_sound = False

pygame.mixer.Sound.play(self.color\_sound)

def play1Menu\_B(self):

#The user has selected to play against the AI.

#Allow the user to play as white or black:

#self.screen.blit(self.playwhite\_pic,(0,self.square\_height\*2))

#self.screen.blit(self.playblack\_pic,(self.square\_width\*4,self.square\_height\*2))

self.selectDifficulty = pygame.image.load('newMedia/selectDifficulty.png')

self.selectDifficulty = pygame.transform.scale(self.selectDifficulty, (350, 80))

self.Easy = pygame.image.load('newMedia/Easy.png')

self.Easy = pygame.transform.scale(self.Easy, (180, 180))

self.Medium = pygame.image.load('newMedia/Medium.png')

self.Medium = pygame.transform.scale(self.Medium, (180, 180))

self.Hard = pygame.image.load('newMedia/Hard.png')

self.Hard = pygame.transform.scale(self.Hard, (180, 180))

self.diffPage.blit(self.box,(0,0))

self.diffPage.blit(self.selectDifficulty, (425-275, 80-15))

self.diffPage.blit(self.Easy, (309-275, 250-15))

self.diffPage.blit(self.Medium, (509-275, 250-15))

self.diffPage.blit(self.Hard, (709-275, 250-15))

self.screen.blit(self.diffPage, (0, 0))

self.diffMenu = 0

global play\_sound

if play\_sound:

play\_sound = False

pygame.mixer.Sound.play(self.difficulty\_sound)

def play2Menu(self):

#The user has selected to play with a friend.

#Allow choice of flipping the board or not flipping the board:

#self.screen.blit(self.flipDisabled\_pic,(0,self.square\_height\*2))

#self.screen.blit(self.flipEnabled\_pic,(self.square\_width\*4,self.square\_height\*2))

self.selectflip = pygame.image.load('newMedia/Flip.png')

self.selectflip = pygame.transform.scale(self.selectflip, (350, 80))

self.enableflip = pygame.image.load('newMedia/enableFlip.png')

self.enableflip = pygame.transform.scale(self.enableflip, (250, 250))

self.disableflip = pygame.image.load('newMedia/disableFlip.png')

self.disableflip = pygame.transform.scale(self.disableflip, (250, 250))

self.flipPage.blit(self.box,(0,0))

self.flipPage.blit(self.selectflip, (425-275, 80-15))

self.flipPage.blit(self.enableflip, (325-275, 280-15))

self.flipPage.blit(self.disableflip, (625-275, 280-15))

self.screen.blit(self.flipPage, (0, 0))

global play\_sound

if play\_sound:

play\_sound = False

pygame.mixer.Sound.play(self.flip\_sound)

def selectMenu(self):

self.selectmode = pygame.image.load('newMedia/selectMode.png')

self.selectmode = pygame.transform.scale(self.selectmode, (350, 80))

self.bymouse = pygame.image.load('newMedia/controlMouse.png')

self.bymouse = pygame.transform.scale(self.bymouse, (250, 250))

self.byvoice = pygame.image.load('newMedia/controlVoice.png')

self.byvoice = pygame.transform.scale(self.byvoice, (250, 250))

self.selectPage.blit(self.box, (0, 0))

self.selectPage.blit(self.selectmode, (425 - 275, 80 - 15))

self.selectPage.blit(self.bymouse, (325 - 275, 280 - 15))

self.selectPage.blit(self.byvoice, (625 - 275, 280 - 15))

self.screen.blit(self.selectPage, (0, 0))

global play\_sound

if play\_sound:

play\_sound = False

pygame.mixer.Sound.play(self.control\_sound)

def call\_board(self):

#All settings have already been specified.

#Draw all the pieces onto the board:

self.drawBoard()

#Don't let the menu ever appear again:

self.isMenu = False

#In case the player chose to play against the AI and decided to

#play as black, call upon the AI to make a move:

if self.isAI and self.AIPlayer==0:

colorsign=1

self.bestMoveReturn = []

self.move\_thread = threading.Thread(target = self.a.negamax,

args = (self.position,self.level,-1000000,1000000,colorsign,self.bestMoveReturn,self.openings,self.searched))

self.move\_thread.start()

self.isAIThink = True

for event in pygame.event.get():

#Handle the events while in menu:

if event.type == KEYDOWN:

if event.key == K\_e:

self.gameEnded = True

elif event.type == QUIT:

self.gameEnded = True

#Window was closed.

#self.gameEnded = True

pygame.mixer.Sound.play(self.exit\_sound)

break

if event.type == MOUSEBUTTONUP:

self.onClick()

def onClick(self):

global play\_sound

#The mouse was clicked somewhere.

#Get the coordinates of click:

posx, posy = pygame.mouse.get\_pos()

if self.buttons[1][0] < posx < self.buttons[1][0] + self.buttons[1][2]:

if self.buttons[1][1] < posy < self.buttons[1][1] + self.buttons[1][3] and self.isAI == -1 :

self.isAI = True

posx , posy = (0 , 0)

if self.buttons[2][0] < posx < self.buttons[2][0] + self.buttons[2][2] :

if self.buttons[2][1] < posy < self.buttons[2][1] + self.buttons[2][3] and self.isAI == -1:

self.isAI = False

posx, posy = (0, 0)

if self.buttons[3][0] < posx < self.buttons[3][0] + self.buttons[3][2]:

if self.buttons[3][1] < posy < self.buttons[3][1] + self.buttons[3][3]:

if self.isAI == True:

if self.diffMenu == -1:

self.AIPlayer = 0

self.isFlip = False

self.diffMenu = 1

posx, posy = (0, 0)

play\_sound=True

elif self.isAI == True and self.select == 1:

self.select = 2

self.temp = 1

posx, posy = (0, 0)

print("Mouse Operated")

elif self.isAI == False:

self.isFlip = True

self.temp = -1

print("Mouse operated")

posx, posy = (0, 0)

if self.buttons[4][0] < posx < self.buttons[4][0] + self.buttons[4][2]:

if self.buttons[4][1] < posy < self.buttons[4][1] + self.buttons[4][3]:

if self.isAI == True:

if self.diffMenu == -1:

self.AIPlayer = 1

self.isFlip = False

self.diffMenu = 1

posx, posy = (0, 0)

play\_sound = True

elif self.isAI == True and self.select == 1:

self.select = 3

self.temp = 1

posx, posy = (0, 0)

print("Voice Operated")

elif self.isAI == False and self.select == 1:

self.isFlip = True

self.select = 3

self.temp = 1

posx, posy = (0, 0)

print("voice Operated")

if self.buttons[5][0] < posx < self.buttons[5][0] + self.buttons[5][2]:

if self.buttons[5][1] < posy < self.buttons[5][1] + self.buttons[5][3]:

self.level = 1

self.select = 1

posx, posy = (0, 0)

play\_sound = True

if self.buttons[6][0] < posx < self.buttons[6][0] + self.buttons[6][2]:

if self.buttons[6][1] < posy < self.buttons[6][1] + self.buttons[6][3]:

self.level = 2

self.select = 1

posx, posy = (0, 0)

play\_sound = True

if self.buttons[7][0] < posx < self.buttons[7][0] + self.buttons[7][2]:

if self.buttons[7][1] < posy < self.buttons[7][1] + self.buttons[7][3]:

self.level = 3

self.select = 1

posx, posy = (0, 0)

play\_sound = True

if self.buttons[8][0] < posx < self.buttons[8][0] + self.buttons[8][2]:

if self.buttons[8][1] < posy < self.buttons[8][1] + self.buttons[8][3]:

self.gameEnded = True

pygame.mixer.Sound.play(self.exit\_sound)

posx, posy = (0, 0)

def Thinking(self):

####while AI is thinking we will cause some fancy movements on screen

self.ax+=1

if self.ax==8:

self.ay+=1

self.ax=0

if self.ay==8:

self.ax,self.ay=0,0

if self.ax%4==0:

self.createShades([])

#If the AI is white, start from the opposite side (since the board is flipped)

if self.AIPlayer==0:

self.listofShades.append(Shades(self.greenbox\_image,(7-self.ax,7-self.ay)))

else:

self.listofShades.append(Shades(self.greenbox\_image,(self.ax,self.ay))) GUI()

# 10.CONCLUSION:

The main focus of this chapter is on the work has done and furture en-hancement of the project. In this project we done the chess game. This game developed using python libraries which also handles the AI. Now we played the chess game using peripheral devices like mobile and voice commans using micro phones.

1. Detecting the position and type of the chess piece.

2. Showing the valid destinations for the current chess piece

3. Displaying the real situation of the chess game in computer screen.

**11. TEAM MEMBERS:**

1. **S.Geethanjali-CSE (731318104006)**
2. **S.Malini- CSE (731318104018)**
3. **S.Maragatham- CSE (731318104019)**
4. **V.Pavithra- CSE (731318104028)**
5. **B.Ponnarasu- CSE (731318104029)**
6. **S.Praveenkumar- CSE (731318104031)**
7. **P.Savitha- CSE (731318104038)**
8. **P.Soundharya- CSE (731318104040)**
9. **M.Ruthramoorthi- CSE (731317104026)**