**Item's In RED TEXT are for assignment 3**

**Key detail about Python:**

Unlike many Java or C like languages the notion of public and private variables is not defined or laid out in in stone. Variables in python are instead given a inferred scope based on where they are initialized. For example, a python function initialized inside a function/method is only accessible to that function and its scope is limited within that function. However in python this is not explicitly stated. In addition objects that are returned also have a larger scope than those private variables simply only declared in the function.

## All of Assignment One's Internal Declarations (MID)

### *Interface: Classes.py*

| Class Name | Functions | Fields | Definition/Origin |
|---|---|---|---|
| Tile | __init__<br>initialization of major fields | index | Index value that represents the type of array in which the Tile may use,single, double, tuple, etc |
| | | tile_size | The actual size of the array, holding integer values |
| | | colour | Simply an RGB variable or Python's colour variables, will represent the colour of a tile |
| | | occupant | Boolean, represents the occupancy of a tile |
| | Init, similar to __init__ but lacking parameters | index | Index value that represents the type of array in which the Tile may use,single, double, tuple, etc |
| | | rect | Stores in array of rectangles for tiles |
| | | tile_size | The actual size of the array, holding integer values |
| | IsOccupied<br>checks for occupancy | occupant | If occupant contains something, true, else none false |
| | __init__<br>initialization of fields | index | Index value that represents the type of array in which the Tile may use,single, double, tuple, etc |
| | | team | Contain whether the piece belongs to the computer player or the human player.(what |

| | | | |
|---|---|---|---|
| **Piece** | | | colour the piece is) integer one or zero, one for computer, zero for human |
| | | ptype | Determines which type of piece your are a normal piece or a king piece. Zero for normal, one for king. |
| | | state | used for drag and drop mechanic. |
| | update<br><br>Updates the changes for the individual checkers pieces | team | Contain whether the piece belongs to the computer player or the human player.(what colour the piece is) integer one or zero, one for |
| | | index | Index value that represents the type of array in which the Tile may use,single, double, tuple, etc |
| | | ptype | Determines which type of piece your are a normal piece or a king piece. Zero for normal, one for king. |
| **Board** | __init__<br>initialization of fields | tile_matrix | Will contain individual indexes for each tile later to represent occupancy of a tile/space on the board |
| | | piece_matrix | Will contain the location of the board and the type of pieces that occur on the board. Also used to count how many pieces are left/ allowed to placed on the board. |
| | Update<br>updates the matrix with new placements on the board or changes to the board. | Will go through both tile_matrix and piece matrix and update the respective fields for the board. This includes, boards, own matrices(tile and piece) and occupancies. | |
| | remove<br><br>removes a checkers piece. | piece | Used to access piece_matrix. |

| | | |
|---|---|---|
| Move<br><br>checks if input from a user is legal and if so, executes the following input. | old_index | Memory state that stores the old index of the moved pieces. |
| | new_index | Updates the new index of the moved pieces. |
| | moving_piece | Checks the type of the piece that is moved with boolean values, 0 for the normal piece, 1 for king piece. |
| | moving_distance | Checks if the move is valid with the distance(slots) from the board that piece has been updated. |
| | kill_distance | 2 element array halving the elements of moving_distance. |
| | kill_index | Index of tile that is potentially being killed |
| | kill_tile | Tile object for the index involved with kill moves. |
| | kill_tile.isOccupied | Checks if there is target piece on specific tile, that is possibly killed. |

A field in python is a variable that may define some properties of that class
A rectangle is created with methods from pygame, which is called Rect, the function call takes the form of Rect(left, top, width, height).

## *Module: Main.py*

| | Name | Definition/Origin |
|---|---|---|
| **Variables** | win | The User window, Global Variable, Origins in screen.py |
| | board | Represents the board, Origins in gameloop.py |

## *Module: screen.py- simply creates the window*

| | Name | Definition/Origin |
|---|---|---|
| **Variables** | window | The User window, Global Variable, Origins in PyGame. |

## *Module: draw.py*

| | Name | Value | Type |
|---|---|---|---|
| **Constants** | darkgrey | (100,100,100) | RGB values |
| | white | (255,255,255) | RGB values |
| | black | (0,0,0) | RGB values |
| | red | (255,0,0) | RGB values |
| | blue | (0,0,255) | RGB values |
| | yellow | (255,255,0) | RGB values |
| | button3Atext | ("force start", True, white) | String |
| | buttonRtext | ("reset", True, white) | String |
| | button1 | ([410, 5], [130, 40], (100,100,100)) | Rectangle(shape) |
| | button2 | ([410, 55], [130, 40],(100,100,100)) | Rectangle(shape) |
| | button1K | ([410, 105], [130, 40], (100,100,100)) | Rectangle(shape) |
| | button2K | ([410, 155], [130, 40],(100,100,100)) | Rectangle(shape) |
| | button3 | ([410, 205], [130, 40],(100,100,100)) | Rectangle(shape) |
| | button3A | ([410, 255], [130, 40],(100,100,100)) | Rectangle(shape) |
| | buttonReset | ([410, 305], [130, 40],(100,100,100)) | Rectangle(shape) |
| | grey | (200,200,200) | RGB values |
| | darkred | (180, 0, 0) | RGB values |
| | darkblue | (0,0,180) | RGB values |
| | PvPtext | font.render("Player vs Player", True, white) | String |
| | BPvCtext | font.render("Blue Player vs CPU", True, white) | String |
| | RPvCtext | font.render("Red Player vs CPU", True, white) | String |
| | PvPbutton | ([100, 50], [350, 95]) | Rectangle(shape) |
| | BPvCbutton | ([100, 150], [350, 95]) | Rectangle(shape) |
| | RPvCbutton | ([100, 250], [350, 95]) | Rectangle(shape) |

| | Name | Definition/Origin |
|---|---|---|
| **Variables** | board | Contains the matrix of the real life checkers board holding both the information on which tile is occupied and how many pieces remain. |
| | tile | Temporary loop variable |
| | piece | Temporary loop variable |
| | button1text | String Function Output (colour change) |

| | Name | Definition/Origin |
|---|---|---|
| **Variables** | button2text | String Function Output (colour change either yellow or white, yellow for pressed) |
| | button1Ktext | String Function Output (colour change either yellow or white, yellow for pressed) |
| | Button2Ktext | String Function Output (colour change either yellow or white, yellow for pressed) |
| | button3text | String Function Output (colour change either yellow or white, yellow for pressed) |
| | **Name** | **Type** |
| **State data** | b1 | Boolean |
| | b2 | Boolean |
| | b3 | Boolean |

## *Module: gameloop.py*

| | Name | Value | Type |
|---|---|---|---|
| **Constants** | grey | (200,200,200) | RGB values |
| | white | (255,255,255) | RGB values |
| | red | (255,0,0) | RGB values |
| | font | ('arial', 14) | FONT |
| | Ecantplacehere | ("CANT PLACE HERE", True, red) | String |
| | Eoutofpieces | ("OUT OF PIECES", True, red) | String |

| | Name | Definition/Origin |
|---|---|---|
| **Variables** | piece_matrix | Private integer Array to be fed later into board module when it is initialized |
| | tile_matrix | Private integer Array to be fed later into board module when it is initialized |
| | board | Global update variable of class type Board, where the Board is represented by matrices and other behaviours. Origin is in the classes module. |
| | unplaced_player_pieces | Private Pieces Class Type, used for storing the number of pieces, private as it is only required for the placement of pieces. Origin the classes module |
| | unplaced_computer_pieces | Private Pieces Class Type, used for storing the number of pieces, private as it is only required for the placement of pieces. Origin is the Classes module |
| | TempC /TempP | Private classes piece, removed piece, used and aliased such that we change the individual type of piece. Origin the classes module |
| | mpos | Private Mouse X and Y coordinate. Pygame function |
| | **Name** | **Type** |
| **State Variables** | click_index | Integer Array |

| | | |
|---|---|---|
| | event | Boolean |
| | placing | Boolean |
| | place_error | Boolean |
| | piece_error | Boolean |
| | placing_player_piece | Boolean |
| | placing_computer_piece | Boolean |
| | placing_player_king | Boolean |
| | placing_computer_king | Boolean |
| | occupied | Boolean |

## *Module:save.py*

| Constants | Name | Value | Type |
|---|---|---|---|
| | file_name | "saves\save1.txt" | string |

| Variables | Name | Definition/Origin |
|---|---|---|
| | index | linear value of index for using when referring to data from piece_list recovered from save-file. |
| | piece_matrix | Used to visit all pieces individually with for loop. |
| | piece_string | String to indicate different types of pieces by concatenating defined string combination. |

| Functions | Fields | Definition/Origin |
|---|---|---|
| **loadGame** | save_file | Open saved file. |
| | piece_string | Get the information from the file. |
| | piece_list | Remove all the "-" for in the data to make the strings usable and simplify. |
| | piece_matrix | initialize the piece matrix as being empty. |
| **saveGame** | save_file | Open a file to save progress of a game. |

## Assignment One's Major Logic Components and Implementation

Since that a major component of the games logic is handled in the application's controller and model , i.e the gameloop and draw module , only these components have changing variables, and only the methods responsible for those changes will require a tabular expression.

## Module draw.py

**Method:drawBoard**

**Input: Win, Board**
**Updates: Win, Board**

**Consists of two parts:**

1. Draw the boards tiles , alternating between black and white. All the while using a nested loop and a matrix
2. Goes through the Piece Matrix and renders the pieces that currently exist in the matrix and draws it onto the board.

**Method: drawButtons**

**Input: Win, Board**
**Updates: Win, Board**

**Consists of two Parts:**

**1.**

| | **Rendering of button text** |
|---|---|
| b1 | button1text = font.render("place RED piece", True, yellow) |
| !b1 | button1text = font.render("place RED piece", True, white) |
| b2 | button2text = font.render("place BLUE piece", True, yellow) |
| !b2 | button2text = font.render("place BLUE piece", True, white) |
| b3 | button1Ktext = font.render("place RED king", True, yellow) |
| !b3 | button1Ktext = font.render("place RED king", True, white) |

| | |
|---|---|
| b4 | button2Ktext = font.render("place BLUE king", True, yellow) |
| !b4 | button2Ktext = font.render("place BLUE king", True, white) |

2. Drawing of button shapes and button text

## Module: gameloop.py

**Method: __run__**
**inputs:win, board**
**updates:event**
simply takes in win and board and checks for events

**Method: placePieces**
**inputs:win, board**
**updates:event,board**

Note that this is a vertical tabular expression , similar to the one in slide 30

This method can be divided into three parts:

<table>
<tr><td colspan="8">event.type== pygame.MOUSEBUTTONDOWN</td></tr>
<tr><td colspan="7">click_index[0]>7</td><td></td></tr>
<tr><td>click_index[1] < 1</td><td>click_index[1] < 2</td><td>click_index[1] < 3</td><td>click_index[1] < 4</td><td>click_index[1] < 5</td><td>click_index[1] < 6</td><td>click_index[1] < 7</td><td></td></tr>
<tr><td>placing_player_piece</td><td>placing_computer_piece</td><td>placing_player_king</td><td>placing_computer_king</td><td>CreateDefaultStart<br><br>placing = False</td><td>placing = False</td><td>Set all variables to zero</td><td></td></tr>
</table>

| |
|---|
| !(click_index[0]>7) |

<table>
<tr><td colspan="3">placing_player_piece==true</td><td colspan="3">placing_computer_piece==true</td><td colspan="3">placing_player_king:==true</td><td colspan="3">placing_computer_king:==true</td></tr>
<tr><td colspan="2">(!occupied) & (board_location.colour == "dark")</td><td>else</td><td colspan="2">(!occupied) & (board_location.colour == "dark")</td><td>else</td><td colspan="2">(!occupied) & (board_location.colour == "dark")</td><td>else</td><td colspan="2">(!occupied) & (board_location.colour == "dark")</td><td>else</td></tr>
<tr><td>try</td><td>exception</td><td rowspan="2">place_error = True</td><td>try</td><td>exception</td><td rowspan="2">place_error = True</td><td>try</td><td>exception</td><td rowspan="2">place_error = True</td><td>try</td><td>exception</td><td rowspan="2">place_error = True</td></tr>
<tr><td>Board matrix updates with player piece</td><td>piece_error = True</td><td>Board matrix updates with computer piece</td><td>piece_error = True</td><td>Board matrix updates with king player piece</td><td>piece_error = True</td><td>Board matrix updates with computer king piece</td><td>piece_error = True</td></tr>
</table>

## Assignment Two's Major Logic Components and Implementation

With the basic logics implemented in assignment one, new changes has been added to be able to move checkers pieces if the user input is valid according to checkers game's rule. Also, if two different types of checkers pieces overlaps, the previous checkers piece will be killed.

New Module has been added, called save.py, which allows user to save the process of the game and load it later on to start at the saved position of the game.

## Module: classes.py

**Method: move**

**Inputs: moving_piece, new_index**

**Updates: moving_distance**

| Result |
| --- |
| |

<table>
<tr><td colspan="2">new_tile.isOccupied</td><td>False</td></tr>
<tr><td rowspan="2">moving_piece.ptype = 0</td><td>(new_index[1] <= moving_piece.index[1]) & (moving_piece.team == 0)</td><td>print "wrong way"<br><br>return False</td></tr>
<tr><td>(new_index[1] >= moving_piece.index[1]) & (moving_piece.team == 1)</td><td>print "wrong way"<br><br>return False</td></tr>
<tr><td rowspan="3">!new_tile.isOccupied</td><td>new_index[0] == moving_piece.index[0]</td><td>print "not diagonal"<br><br>return False</td></tr>
<tr><td>new_tile.colour == "light"</td><td>print "white tile"<br><br>return False</td></tr>
<tr><td>new_index[0] !== moving_piece.index[0]<br><br>&<br><br>new_tile.colour !== "light"</td><td>moving_distance = [new_index[0] - old_index[0], new_index[1] - old_index[1]]</td></tr>
<tr><td colspan="2">abs(moving_distance[0]) == 1 & (abs(moving_distance[1]) == 1)</td><td>self.remove(moving_piece)<br><br>moving_piece.index = new_index<br><br>self.piece_matrix[new_index[0]][new_index[1]] = moving_piece<br><br>print "legal"<br><br>return True</td></tr>
<tr><td colspan="2">abs(moving_distance[0]) == 2</td><td>kill_distance = [moving_distance[0]/2, moving_distance[1]/2]<br><br>kill_index = [old_index[0] + kill_distance[0], old_index[1] + kill_distance[1]]<br><br>kill_tile = self.tile_matrix[kill_index[0]] [kill_index[1]]<br><br>print kill_index</td></tr>
</table>

| | | |
|---|---|---|
| abs(moving_distance[0]) == 2 | kill_tile.isOccupied() = False | print "no one to kill"<br><br>return False |
| | kill_tile.occupant.team == moving_piece.team: | print "same team"<br><br>return False |
| abs(moving_distance[0])!==2 | | print "good move"<br><br>self.remove(kill_tile.occupant)<br><br>self.remove(moving_piece)<br><br>moving_piece.index = new_index<br><br>self.piece_matrix[new_index[0]][new_index[1]] = moving_piece |

## Module: save.py

**Method: loadGame, saveGame**

**Inputs: board, piece_matrix**

**Updates: board.piece_matrix, save_file**

**#####data in the save-file adheres to the following:#####**

**# 0 = No piece**

**# 1 = Normal red piece**

**# 2 = King red piece**

**# 3 = Normal blue piece**

**# 4 = King blue piece**

**loadGame opens a file the user saved. The file contains saved game's pieces' location on the board with specification of types with unique number, as I noted above. The logic is simple, the load method access the board matrix and go through all the slots and check if there's number(indicating each types of pieces) saved for that slot.**

## Tabular expression for loadGame

| Result | |
|---|---|
| piece_list[index] == '0' | piece_matrix[i][j] = None |
| piece_list[index] == '1' | piece_matrix[i][j] = classes.Piece([i,j], 0, 0) |
| piece_list[index] == '3' | piece_matrix[i][j] = classes.Piece([i,j], 0, 1) |
| piece_list[index] == '2' | piece_matrix[i][j] = classes.Piece([i,j], 1, 0) |
| piece_list[index] == '4' | piece_matrix[i][j] = classes.Piece([i,j], 1, 1) |

Similar to load game, saveGame opens a file to write. The file initially contains empty string, and then access the board matrix and go through the board and saves pieces on the board, by concatenating the unique number identifications to the empty string.

## Tabular expression for saveGame

| Result | | |
|---|---|---|
| piece == None | | piece_string += "0-" |
| piece.team == 0 | piece.ptype == 0 | piece_string += "1-" |
| | piece.ptype == 1 | piece_string += "3-" |
| piece.team == 1 | piece.ptype == 0 | piece_string += "2-" |
| | piece.ptype == 1 | piece_string += "4-" |

## Assignment Three's Major Logic Components and Implementation

New module, named AI.py has been added, which will control all the game logics of the pieces; their movement calculation and turn logics. All possible moves will be recorded into an array and will be used to define random movement of AI. All the events will be handled in gameloop.py, such as moving a piece, execution of AI's random movement, possible kill moves( including chaining) and switching turn after a movement (possibly more than once depends on chaining condition).

There are three possible modes for user, which is Player vs Player, Player (Blue piece) vs AI, and Player( Red piece) vs AI. The event for the mode selection will be handled in the method modeSelect in gameloop.py.

## Module: AI.py

Method: isSame

Uses: move, moving_piece, destination_index

Updates: Conditions

Description: used to check moving_piece == move.moving_piece, destination_index == move.new_index and returns the condition.

| Result | | |
|---|---|---|
| moving_piece !== move.moving_piece | | False |
| moving_piece == move.moving_piece | destination_index == move.new_index | True |
| | destination_index !==move.new_index | False |

Method: allPossibleMoves

Uses: board, turn

Updates: possible_moves, legal_moves

| Result |
|---|

| | |
|---|---|
| piece == True | new_moves = possibleMoves(piece, board, turn)<br>legal_moves = legal_moves + new_moves |
| legal_moves[i].kill != None | possible_moves.append(legal_moves[i]) |
| len(possible_moves) == 0 | possible_moves = legal_moves |

Method: possibleMoves

Uses: piece, board, turn

Updates: legal_moves, piece_index

Description: Method that will return an array of all possible moves.

| **Result** | | |
|---|---|---|
| ((i == 0) & (j == 0)) | | Continue |
| ((row < 0) \| (row > 7) \| (column < 0) \| (column > 7)) | | Continue |
| board.tile_matrix[row][column].colour == "light" | | Continue |
| !( ((i == 0) & (j == 0)) &<br>((row < 0) \| (row > 7) \| (column < 0) \| (column > 7)) &<br>board.tile_matrix[row][column].colour == "light") | | legal_move = checkMove(board, piece, [row, column], turn) |
| !( ((i == 0) & (j == 0)) &<br>((row < 0) \| (row > 7) \| (column < 0) \| (column > 7)) & board.tile_matrix[row][column].colour == "light") | legal_move == True | legal_moves.append(legal_move) |
| | legal_move == False | continue |

Method: checkMove

Uses: board, moving_piece, new_index, turn

Updates: old_index, piece_type, new_tile

Description: Method that checks if move is legal and executes the valid moves.

<table>
<tr><th colspan="4">Result</th></tr>
<tr><td colspan="3">if moving_piece.team != turn:</td><td>return None</td></tr>
<tr><td colspan="3">new_tile.isOccupied() == True</td><td>return None</td></tr>
<tr><td rowspan="9">new_tile.isOccupied() == False</td><td>moving_piece.ptype == 0</td><td>(new_index[1] <= moving_piece.index[1]) & (moving_piece.team == 0)</td><td>return None</td></tr>
<tr><td>moving_piece.ptype !== 0</td><td>(new_index[1] >= moving_piece.index[1]) & (moving_piece.team == 1)</td><td>return None</td></tr>
<tr><td colspan="2">new_index[0] == moving_piece.index[0]</td><td>return None</td></tr>
<tr><td colspan="2">new_tile.colour == "light"</td><td>return None</td></tr>
<tr><td colspan="2">moving_piece.ptype !== 0 & new_index[0] !== moving_piece.index[0] & new_tile.colour !== "light"</td><td>moving_distance = [new_index[0] - old_index[0], new_index[1] - old_index[1]]</td></tr>
<tr><td rowspan="2">moving_piece.ptype !== 0 & new_index[0] !== moving_piece.index[0] & new_tile.colour !== "light"</td><td>(abs(moving_distance[0]) == 1) & (abs(moving_distance[1]) == 1)</td><td>return Move(moving_piece, new_index, None)</td></tr>
<tr><td>abs(moving_distance[0]) == 2</td><td>kill_distance = [moving_distance[0]/2, moving_distance[1]/2]<br><br>kill_index = [old_index[0] + kill_distance[0], old_index[1] + kill_distance[1]]</td></tr>
</table>

| | | | | |
|---|---|---|---|---|
| | | | | kill_tile = board.tile_matrix[kill_index[0]][kill_index[1]] |
| | | abs(moving_distance[0]) == 2 | kill_tile.isOccupied() != True | return None |
| | | | kill_tile.occupant.team == moving_piece.team | return None |
| | | abs(moving_distance[0]) !== 2 | | return Move(moving_piece, new_index, kill_tile.occupant) |

## Module:gameloop.py

Method: __run__

Uses: win, board, game_mode

Updates: display, board

This method will handle turn logic for computer, events, move logic, win condition and updates the board.

| Result | |
|---|---|
| game_mode == "BPvC" | cpu_team = 0 |
| game_mode == "RPvC" | cpu_team = 1 |
| game_mode !== "BPvC" & game_mode !== "RPvC" | cpu_team = 2<br><br>possible_moves = AI.allPossibleMoves(board, board.turn) |
| (cpu_team == board.turn) | pygame.time.wait(250) |

<table>
<tr><td colspan="3"></td><td>ran = random.random()</td></tr>
<tr><td>(cpu_team == board.turn)</td><td colspan="2">(turnCounter != 0)</td><td>ran_seed = int(ran*len(possible_moves))<br><br>ran_move = possible_moves[ran_seed]<br><br>moving_piece = ran_move.moving_piece<br><br>new_index = ran_move.new_index<br><br>moving = True</td></tr>
<tr><td rowspan="3">event.type == pygame.MOUSEBUTTONDOWN</td><td>mouse_index[0] > 7</td><td>mouse_index[1] < 1</td><td>save.saveGame(board.piece_matrix)<br><br>pygame.quit()<br><br>sys.exit()</td></tr>
<tr><td colspan="2">mouse_index[0] <= 7</td><td>holding = True<br><br>selected_piece = board.piece_matrix[mouse_index[0]][mouse_index[1]]</td></tr>
<tr><td>mouse_index[0] <= 7</td><td>selected_piece != None</td><td>selected_piece.state = "active"</td></tr>
<tr><td colspan="3">event.type == pygame.MOUSEBUTTONUP</td><td>holding = False</td></tr>
<tr><td>event.type == pygame.MOUSEBUTTONUP</td><td colspan="2">mouse_index[0] <= 7</td><td>moving_piece = selected_piece</td></tr>
</table>

<table>
<tr><td colspan="3"></td><td colspan="2"></td><td>new_index = mouse_index<br><br>moving = True</td></tr>
<tr><td colspan="5">event.type == pygame.QUIT</td><td>running = False</td></tr>
<tr><td rowspan="6">moving == True</td><td colspan="4">AI.isSame(move, moving_piece, new_index)</td><td>board.move(move)</td></tr>
<tr><td colspan="3">AI.isSame(move, moving_piece, new_index)</td><td>move.kill == True</td><td>chain_moves = AI.possibleMoves(move.moving_piece, board, board.turn)<br><br>possible_moves = []</td></tr>
<tr><td rowspan="4">AI.isSame(move, moving_piece, new_index)</td><td rowspan="4">move.kill == True</td><td>move.kill == True</td><td>possible_moves.append(move)</td></tr>
<tr><td>len(possible_moves)>0</td><td>chaining = True</td></tr>
<tr><td>len(possible_moves)<=0</td><td>chaining = False<br><br>turnCounter += 1</td></tr>
<tr><td>len(possible_moves)<=0</td><td>board.turn == 1</td><td>board.turn = 0</td></tr>
<tr><td colspan="2"></td><td>len(possible_moves)<=0</td><td>board.turn == 0</td><td>board.turn = 1</td></tr>
</table>

| | | | | | |
|---|---|---|---|---|---|
| | | | move.kill == False | | turnCounter += 1 |
| | | | move.kill == False | board.turn == 1 | board.turn = 0 |
| | | | | board.turn == 0 | board.turn = 1 |
| chaining == False | | | | | possible_moves = AI.allPossibleMoves(board, board.turn) |
| len(possible_moves) == 0 | | | board.turn == 0 | | winning_team = "blue" |
| | | | board.turn == 1 | | winning_team = "red" |
| | | | turnCounter == 0 | | winning_team = "invalid"<br>draw.winScreen(win, winning_team, big_font,turnCounter)<br>pygame.time.delay(2000)<br>pygame.quit() |

Method: modeSelect

Uses: win

Updates: display

Description: This method will select which mode player will be playing.

| **Result** | | |
|---|---|---|
| event.type == pygame.MOUSEBUTTONDOWN | PvPbutton.collidepoint(pygame.mouse.get_pos()) | return "PvP" |
| | BPvCbutton.collidepoint(pygame. | return "BPvC" |

| | | |
|---|---|---|
| | mouse.get_pos()) | |
| | RPvCbutton.collidepoint(pygame.mouse.get_pos()) | return "RPvC" |
| event.type == pygame.QUIT | | pygame.quit()<br>sys.exit() |

**Module:draw.py**

Method: winScreen

Uses: win, team, font, turnCounter

Updates: display

Description: This method will display winning screen when one side wins a game or invalid game start has occurred (by meaning invalid, when user press force start button on an empty board).

| **Result** | | |
|---|---|---|
| team == "invalid" | | raw_text = "A first turn can't be played! :O"<br>color = darkred |
| team !== "invalid" | | raw_text = "The " + team + " player wins in "+ str(turnCounter)+ " move(s)!" |
| team !== "invalid" | team == "blue" | color = blue |
| | team == "red" | color = red |