Program Design Methods and Intro to Programming Python Final Project: Shogi Sim



**Student Information:**

Morris Kim / 2440074174

**Class Information:**

Class: L1BC  
Lecturer: Ida Bagus Kerthyayana / D5757 (might be wrong)

**Binus University International  
School of Computer Science  
2021**

Chapter I – Introduction

When this final project was first introduced to the class, the prospect of being able to make anything we want, limited only by our creativity and time seemed exciting. The only caveat was that an external library must be used. That first night, several hours was spent thinking of feasible and entertaining ideas. During that duration, 3 ideas came to mind: An RTS (Real Time Strategy) game where the player control resources and produce units to defeat an enemy; An chatbot-like AI that one can talk to and hold somewhat coherent conversations with; Shogi.

It should now be noted that although the first 2 ideas were considerably challenging, Shogi was not chosen because it was simpler to do. By the time the project started its development, it was already mid-December. After watching a 7 hour tutorial video on TensorFlow, it was decided to perhaps leave the idea for a future project since there might not be enough time to finish things given my own inexperience with dealing with the subject. The first idea was also questionable when considering the time limit given all the preparation needed in creating the assets, design, sound, and etc.

Hence, in the end, Shogi was chosen. With the popularity of the recent release of Netflix’s historical drama *Queen’s Gambit* and everyone’s recommendation to go and watch it, it is only appropriate to work on a variant of chess that is lesser known to the general audience.

This project is completely open source, and can be accessed through my github repository linked here: <https://github.com/morris-necc/shogi_arcade>

Chapter II – Project Specifications

**Project purpose:**

This project’s purpose is to accurately represent a game of Shogi so that people can play the game with either themselves or other people. Another purpose is to introduce Shogi to more people as a fascinating board game.

**Project audience:**

This project’s audience is individuals who are interested in Shogi, or have played Shogi in the past and are interested to play it again.

**Project aim:**

This project’s aim is to simulate a game of Shogi. Every piece on the board must work like it should in the actual board game, and the mechanics of the game must be replicated exactly.

**Project Requirements:**

* Pieces objects, each with their own set of movements
* A promoted version of each piece, again with their own set of movements
* Calculations to highlight which moves are available for every piece
* Calculations for other mechanics such as pinning, checking, and checkmating
* Accurate sound effects featuring an angry Asian man calling the player a disgrace for losing
* A simple design to make pieces readable for a western audience

Chapter III – Solution Design

**1. Overview**

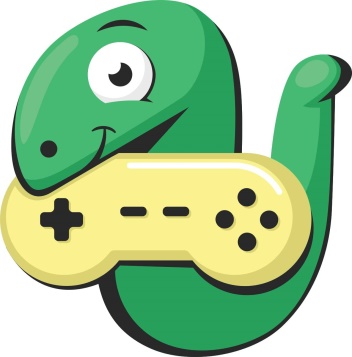
****

Image 1. Arcade logo  
(<https://arcade.academy/_images/arcade-logo.svg>)

*Shogi Sim* uses Python 3.8.1 using arcade 2.5.1 as an external library. It is used to make the window, draw objects on that window, detecting mouse clicks in that window, as well as to load and play the appropriate sounds.

**2. A brief explanation of every piece’s movements**

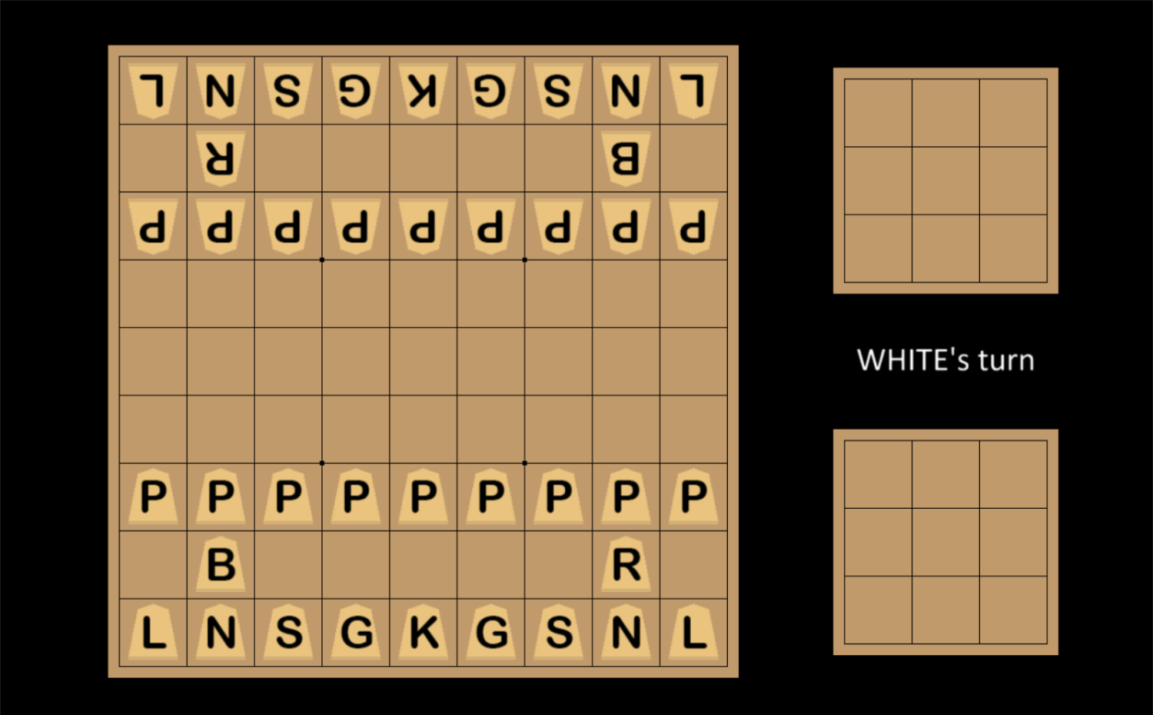


Image 2. The Shogi board

In Shogi, pieces are usually made of wood with a slightly tipped edge not unlike the program’s representation. These pieces usually have kanji characters to represent them, but for ease of understanding, the starting letter for the piece is used instead -- with the exception of the Knight piece (N), so it is not confused with the King.

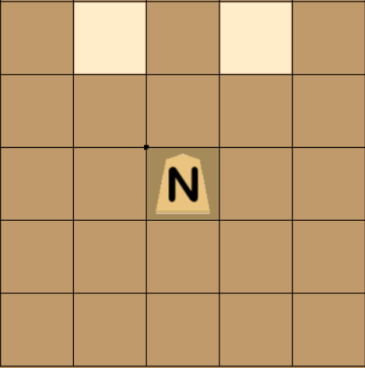
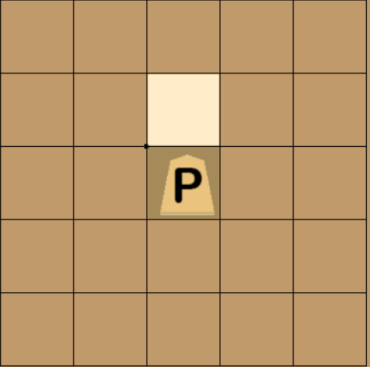


Image 3, 4. The pawn and the knight piece

Although the pawn’s movement in chess and Shogi are similar, pawns in Shogi cannot take on the diagonal and is only able to move vertically in any circumstances. Likewise, the knights in Shogi can only move in a vertical “L” motion and are therefore unable to flexibly jump around the board like its chess counterpart.

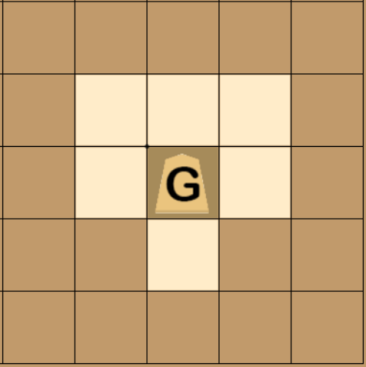
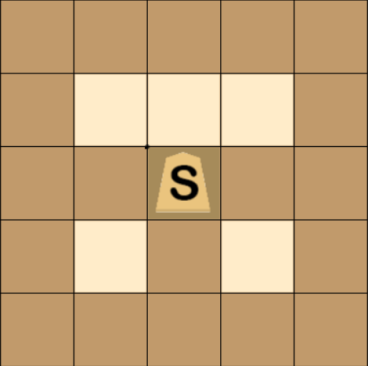
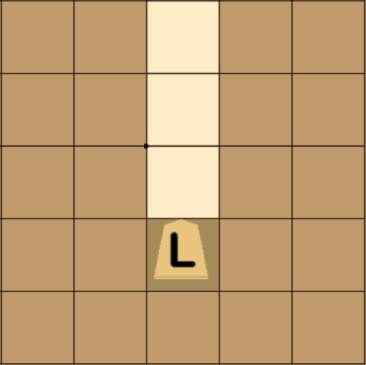


Image 5, 6, 7. The lance, silver general, and gold general

There are 3 new pieces in Shogi: the Lance, Silver General, and Gold General. The lance piece is able to move vertically forwards an indefinite amount of squares until another piece is met; the silver general is able to move similar to the king, but it is unable to move to its sides and the square below it. The gold general, like the silver general, also moves like the king but with the exception of its bottom two corners.

Another major difference is in how pieces promote. In shogi, almost every piece can promote with the caveat of not being able to choose what piece it will be. Fortunately, Promoted versions of each piece are almost always better than the original.

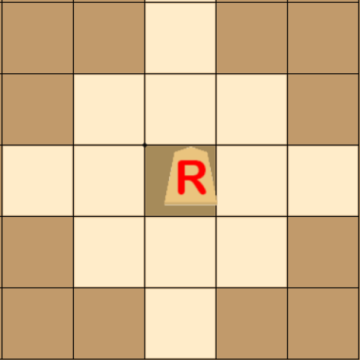
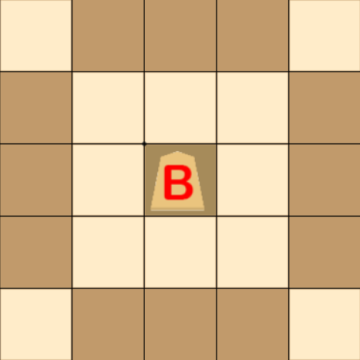
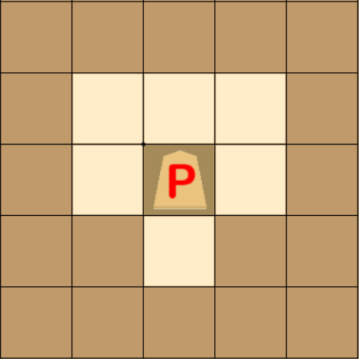


Image 8, 9, 10. The 2 files

The pawn, knight, lance, and silver promote to the same type of movement, while both the bishop and the rook can now also move in a whole square around it (like the king).

**3. File dependency and general structure**

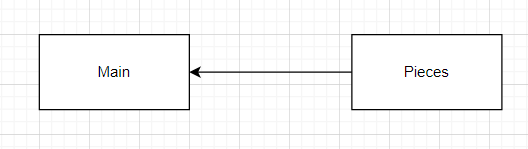
****

Image 11. The 2 files

Disregarding the sound and sprite files, only 2 python files are used in the development of the program. The actual relationship is a little more complicated than this, but this diagram can be used as a general idea of how it works.

As the name suggests, Pieces contain the classes for every single type of piece possible. The main file imports these classes and uses them to perform the logic of the board (which is mostly in the visual side) such as selecting pieces, updating their positions on the board, and giving the promotion prompt when the piece is in range among many other things.

To get a vague sense of what each of these files do, feel free to refer to the class structure diagrams below. MyGameWindow is a class inherited from arcade’s Window class, and it also doubles as the class representing the board, simultaneously giving the visuals and running the game’s logic.

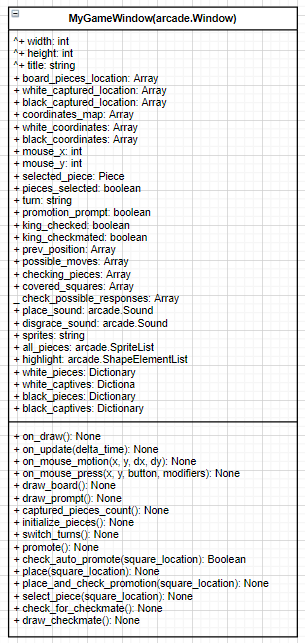


Image 12. MyGameWindow class structure

To explain what every attribute and method of the class does will require an unnecessary length of time, and so they will be separated into categories based on their function. To begin with, here are explanations for all the attributes:

* Width, height, and title have to do with the game window and are inherited directly from the arcade Window class. These attributes change exactly what their names suggests
* Board\_pieces\_location, white\_captured\_location, black\_captured\_location, coordinates\_map, white\_coordinates, black\_coordinates have to do with the board and piece locations within that board. Since there are 3 boards, 3 lists are needed to store Pieces objects in their right location while the other 3 lists are there to help with converting the x and y coordinates of the board to the x and y coordinates of the screen and vice versa
* Mouse\_x, mouse\_y, selected\_piece, pieces\_selected, prev\_position, possible\_moves, and highlight have to do with piece movement. Mouse coordinates and the pieces\_selected attribute helps in showing the players know that they have selected a piece. Selected\_piece, prev\_position, possible\_moves, and highlight helps in figuring out which available square can the selected piece go to (including its current square if the player changes their mind) and shows it by highlighting said squares. The attribute turn keeps track of whose turn it is
* King\_checked, king\_checkmated, checking\_pieces, covered\_squares, and check\_possible\_responses have to do with checking and checkmating. The first 2 are Boolean values that return true when conditions are met. Covered\_squares and check\_possible\_responses are used to calculate these 2 conditions while checkin\_pieces is used to help with calculating mechanics such as pinning and blocking.
* White\_pieces, white\_captives, black\_pieces, and black\_captives are used to initialize pieces to its correct location as well as to implement shogi’s captive system, where the player can place captured opponents back to the board.
* Promotion\_prompt is a unique attribute, since the promotion is the only mechanic that requires a prompt that asks the player whether or not they want to promote. When set to true, the game pauses until the player chooses whether to promote or keep the piece unpromoted.
* Place\_sound and disgrace\_sound are both to play sound in the game. The former is played whenever the player select a piece and put it down while the latter plays only at the end of the game when one player loses.
* Sprites and all\_pieces have to do with showing the pieces on the screen. Sprites is to hold the directory of all the sprites whilst all\_pieces is a SpriteList so that every piece is drawn all at once rather than one at a time.

And here are the explanations for all the methods:

* On\_draw, draw\_board, draw\_prompt, initialize\_pieces, draw\_checkmate, and captured\_pieces\_count are used to draw the visual representations of the board and the pieces on the window. Draw\_prompt and draw\_checkmate is used to cue to the players that an event has happened (promotion and checkmate respectively)
* On\_update, on\_mouse\_motion, and select\_piece are used to both select a piece and show the player that they have selected a piece. Square\_location refers to the square coordinates on the **board** that the last click is located in.
* Promote and check\_auto\_promote is used to promote the piece. There are 3 different types of promoted pieces, and the first method is used to check for which piece\_type the piece is, and pick its promotion according to that. Pawns, knights, and lances will automatically promote if it reaches the edge of the board, where it is unable to move backwards nor forwards
* Place, place\_and\_check\_promotion have to do with placing the pieces down. The latter checks if it is available for promotion. The former is a part of the latter function, but is also used in other locations as well.
* On\_mouse\_press is where most of the game logic happens. This is because the player initiates events by clicking, whether it be selecting a piece, taking a piece, and promoting a piece.

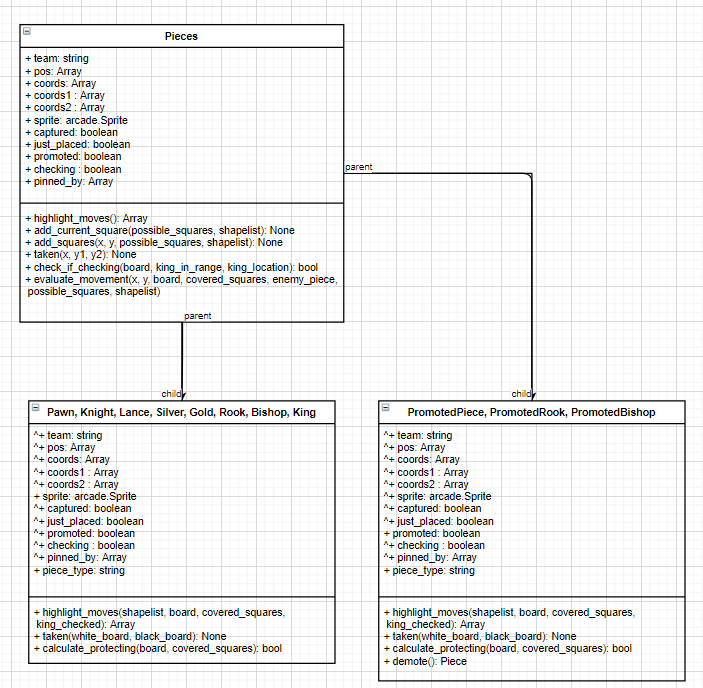


Image 13. Pieces class structure diagram

In order to simplify the diagram, the Pawn, Knight, Lance, Silver, Gold, Rook, Bishop, and King are placed under the same box and PromotedPiece, PromotedRook, and PromotedBishop in another box. They are all mostly similar with the exception of their movement patterns, which are different for every piece.

Another thing to note here is that although King is grouped into with all the unpromoted pieces, it is lacking the taken() method and owns the “checked” attribute. It is placed there as the boxes are quite large and inserting another one will affect the ability to read the diagram without the need to squint.

Explanation of every attribute in the Pieces class and its subclasses:

* Team states what team the piece belongs to. It is always either “black” or “white”
* Pos, coords, coords1, and coords2 refers to the positions the board is on the screen. The x and y values that pos stores refers to the screen’s coordinates while the other 3 refers to the x and y positions on the different boards. Since there are 3 boards, it is split into 3 attributes. This method is used instead of a 2D array since it would be easier to reference the main board’s x and y values individually.
* Captured and just\_placed helps with the capturing and replacing aspect of the game. Just\_placed prevents the prompt from appearing if the player had just placed it back on the board.
* Promoted refers to whether or not the piece is promoted
* Checking refers to whether it is currently checking opposing king
* Pinned\_by refers to whether it is being pinned by a long-range piece (ie. Bishop, rook, and lance)
* Piece\_type is used to refer to its piece type. It is used to ease the process of implementing the capturing and promotion mechanics.

Explanation of every method in the Pieces class and its subclasses:

* Highlight\_moves returns an array consisting of every possible move a piece can make depending on its circumstances as well as highlighting each of those squares. Add\_current\_square and add\_square are helper methods to shorten the code and avoid repetition. The former adds the piece’s current square and shades it into a darker brown while add\_square adds a specified square to the array and highlighting it a lighter colour
* Taken is a method that runs when the piece is taken. The method in the parent class changes its captured attribute to true and shows it on the opposing team’s small board while the child class places it to the right location in the location attributes in the main program
* Check\_if\_checking is a helper method to check if the king is in range of the piece and then set the opposing king’s checked attribute to true via referencing its position on the board array
* Evaluate\_movement simulates whether moving the piece will expose the king to a check. This is done to implement both pinning and blocking. Pinned pieces cannot move to locations that will expose the king while pieces can only move in squares that block the check if the king is checked.
* Calculate\_protecting is done to calculate every single square the piece is covering. Although it sounds similar to the highlight\_moves method, it is used to calculate the piece’s **future** moves since it is done at the end of each turn. It is used in tandem with the check\_if\_checking method to return a Boolean value if the piece is checking the king.

**4. Visually and mechanically representing the board and its pieces**

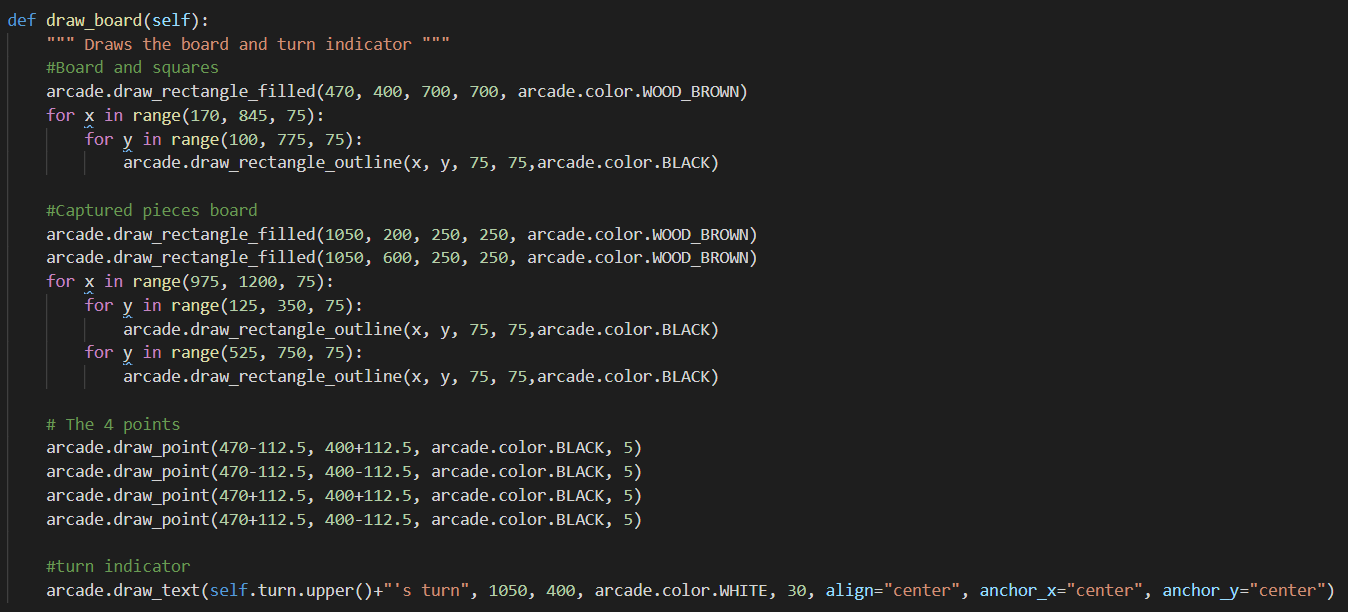
****

Image 14. Drawing the board

Taking advantage of the simple design of the shogi board, this program fully utilizes arcade’s draw functions in order to visualize the board. The board’s background is drawn as a filled rectangle with the preset WOOD\_BROWN color. In addition to that, it might be noted that instead of drawing lines to form the grid, the method of drawing rectangle outlines to form the grid is used instead. This is intended in order to ease development process and make repositioning the grid faster and more efficient. This same process is used to create the smaller boards that are used to keep captured pieces.

The 4 points around the center is calculated with a simple formula of being 1 and a half squares away from the center point of the board where one square has side length of 75 pixels. The turn indicator is put at the side in between the opposing player’s small boards in order to not obstruct the screen.

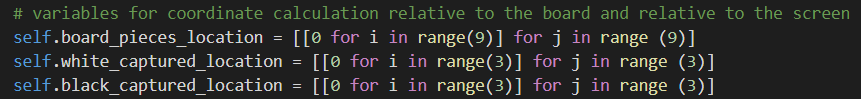
****

Image 15. The actual board

The board is represented in the code as a 2-dimensional array. There are 3 boards, hence the 3 variables. I refrained from making it a 3-dimensional array due to this method being easier to reference and debug.

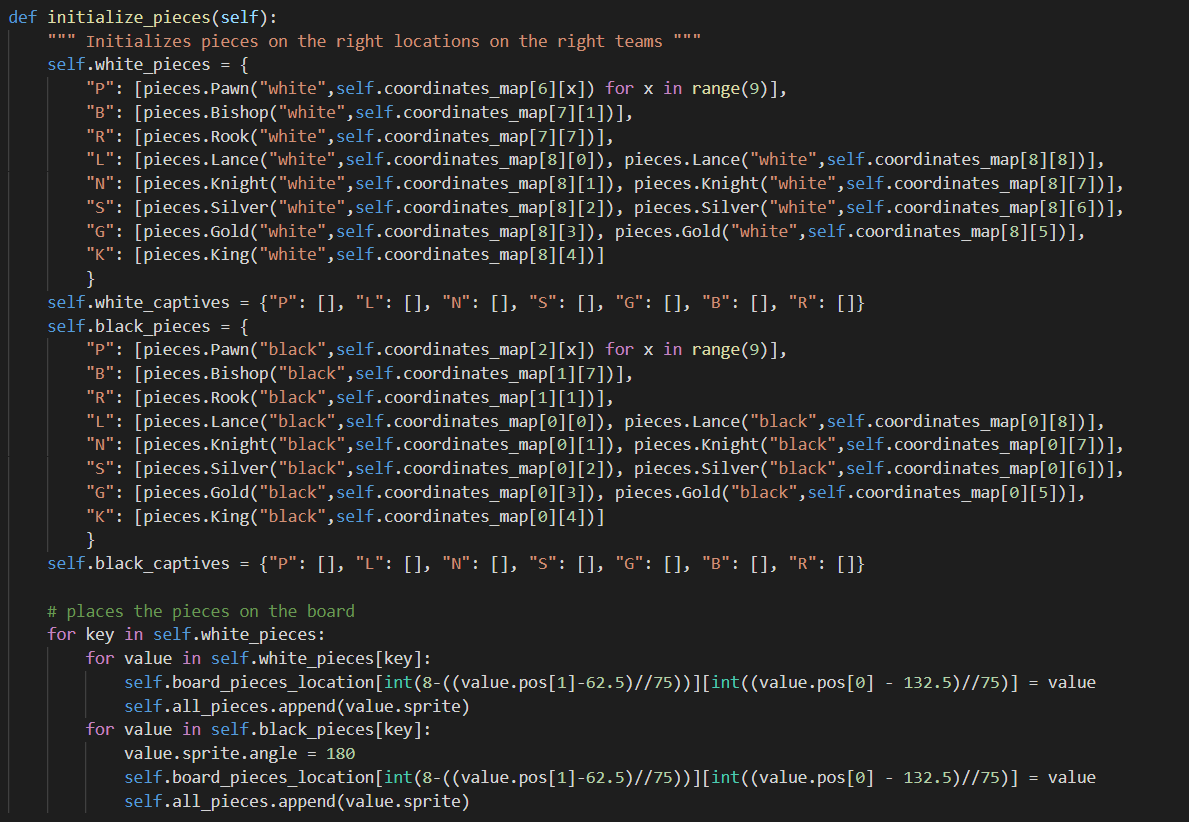
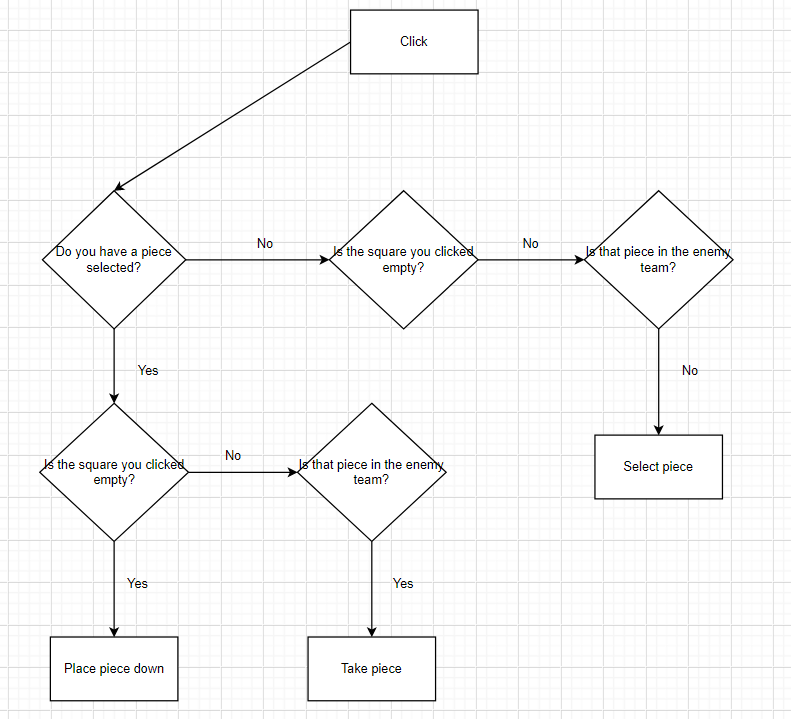
****

Image 16. Drawing and placing the pieces

Like mentioned earlier, the initialize\_pieces method is used to place the pieces on the board. By board, this means both the visual board that the player can see on the screen and the actual board array.

**5. Moving pieces, taking pieces, and placing pieces back down**

****Image 17. A simplified diagram

Any piece movement is done by the logic shown in the above diagram. The diagram is extremely simplified in that it assumes that the player has clicked in a legal location – for example, trying to move a piece to an unhelpful location when the player’s king is in check or trying to drop a captured piece right on top of a non-empty square is considered illegal.

Although easy to understand, the process of implementing just the process was challenging.

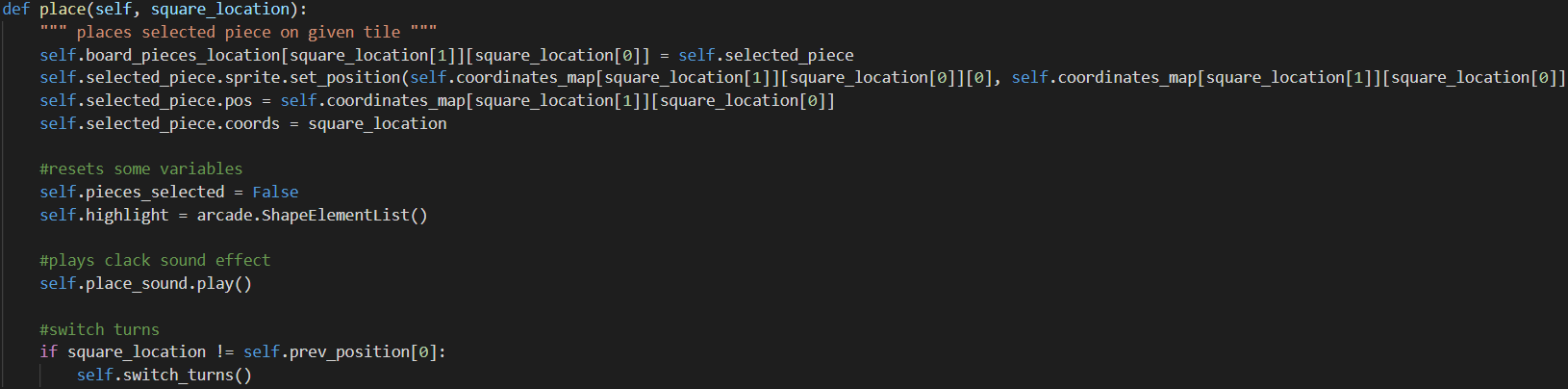


Image 18. The code that runs when the player places a piece down

For example, placing a piece requires the program to update its location on both the visible and invisible board, re-calculate its position relative to the screen and the board, reset the pieces\_selected Boolean, reset the highlights list so the highlights disappear, play the sound effect of placing the piece down, and finally switch turns to the opposite team only if the player has not placed the piece back to its original position.

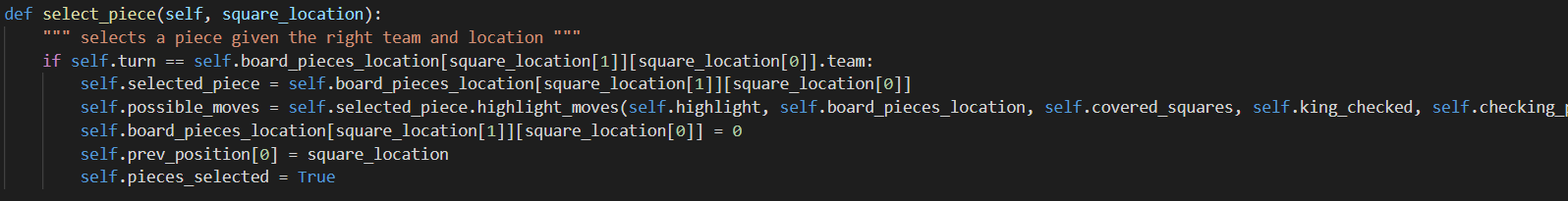


Image 19. The code that runs when the player selects a piece

Likewise, selecting a piece requires the program to set the selected\_piece value to the piece class that the player has selected, calculate and highlight the selected\_piece’s every possible move including its current square, empty its current square on the invisible board as the player is holding it in their hand, set the previous position on the board so that when the player places the piece back to its original place the program would know not to switch turns, and finally set the pieces\_selected Boolean value to True.

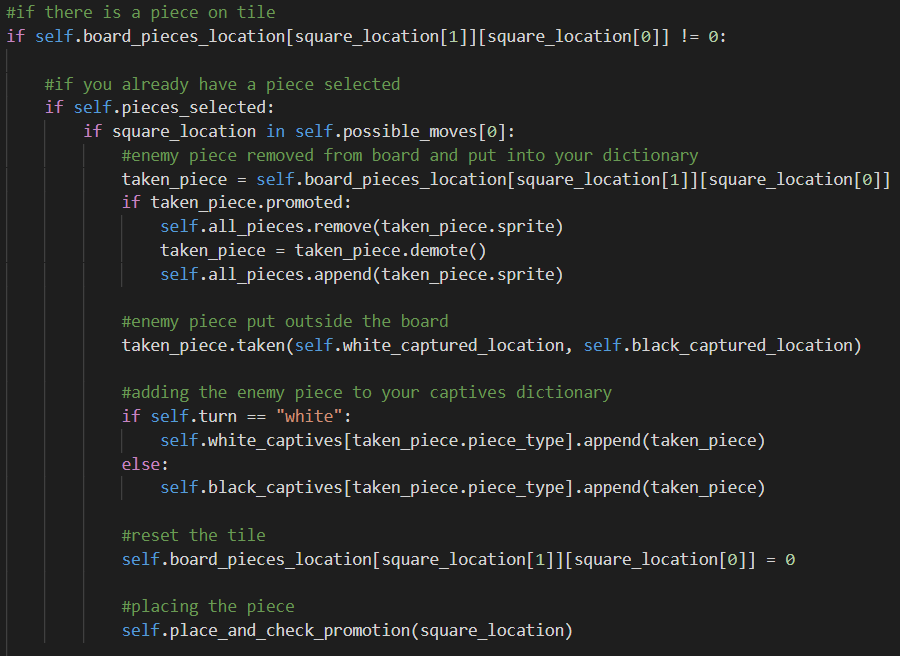


Image 20. The code that runs when the player takes a piece

When taking a piece, it is important to remember that the player is also placing the piece down, hence the place\_and\_check\_promotion method at the bottom of the above image. Most of the code is explained through comments already, but to detail further upon what has already been given: promoted pieces will be demoted when they are taken, and the taken() method of the piece will place it on one of the two small boards depending on the player’s team.

The place\_and\_check\_promotion method is slightly different from the place method and will be covered in the next section regarding promotion.

**6. Promotion**

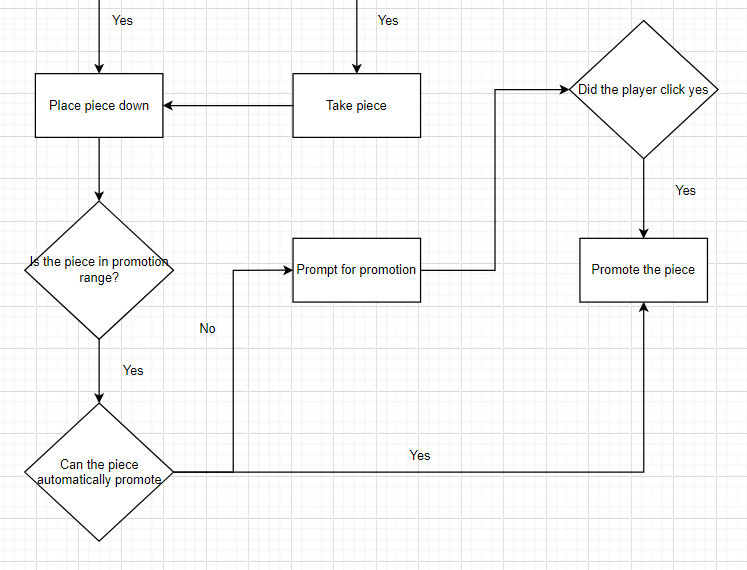
****

Image 21. New and improved logic

Now that placing pieces has been covered, it is time to introduce the concept of promotion. Improving on the previous diagram (image 17), the process of taking pieces is now continued with the process of placing pieces down.

In shogi, promotion does not happen at the opponent’s first row. Instead, it happens on the opponent’s first 3 rows (which will be called the promotion range from now on). In addition to that, pieces such as the pawn, knight, and lance can only move forwards. If these pieces don’t have any possible moves forwards, it will automatically promote.

One thing to note is that whenever a dead end is met (for example, an if condition only branching off into one path), assume that nothing happens on the other path.

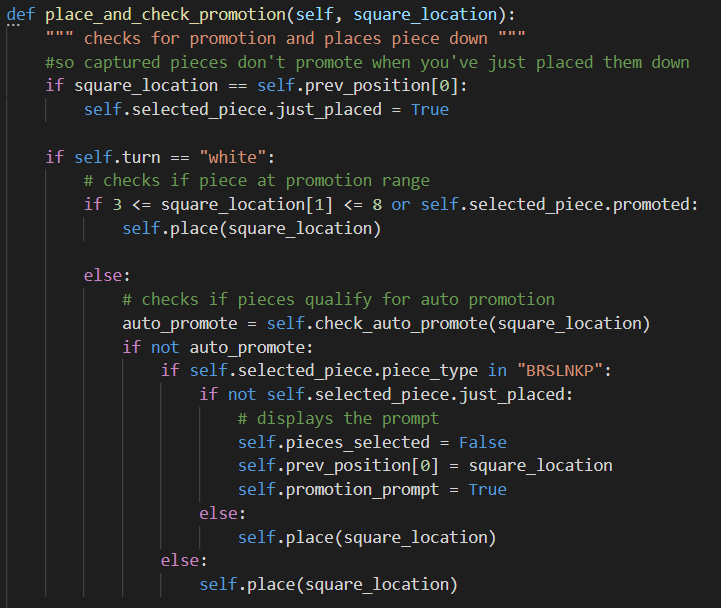
****

Image 22. Placing and checking for promotion

The very first line sets the piece’s just\_placed Boolean value to true if the player clicks on the same square as its previous location. This prevents players being able to promote without switching turns and hence obtain an extra free turn.

Since the two teams have different promotion ranges, it is necessary to split the code into two paths. Because the two paths run on similar logic, only one path is shown here for the sake of brevity.

If the player places the piece outside the promotion range, the program will simply place the piece in that position. Otherwise, it will check for automatic promotion. If the piece does not qualify for automatic promotion, check for the kind of piece the player is moving. If the piece is the kind that is able to promote, it will finally display the prompt for promotion.



Image 23. Promotion prompt

If the player places the piece outside the promotion range, the program will simply place the piece in that position. Otherwise, it will check for automatic promotion. If the piece does not qualify for automatic promotion, check for the kind of piece the player is moving. If the piece is the kind that is able to promote, it will finally display the prompt for promotion.

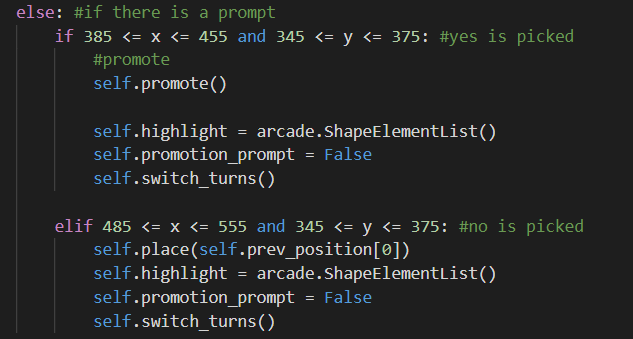


Image 24. When there is a prompt

When a prompt is shown on the screen, the player must click in the “yes” area or the “no” area in order to continue the game. When the player presses “Yes”, the piece will promote and if the player presses “No”, the piece will not promote and will instead be simply placed there.

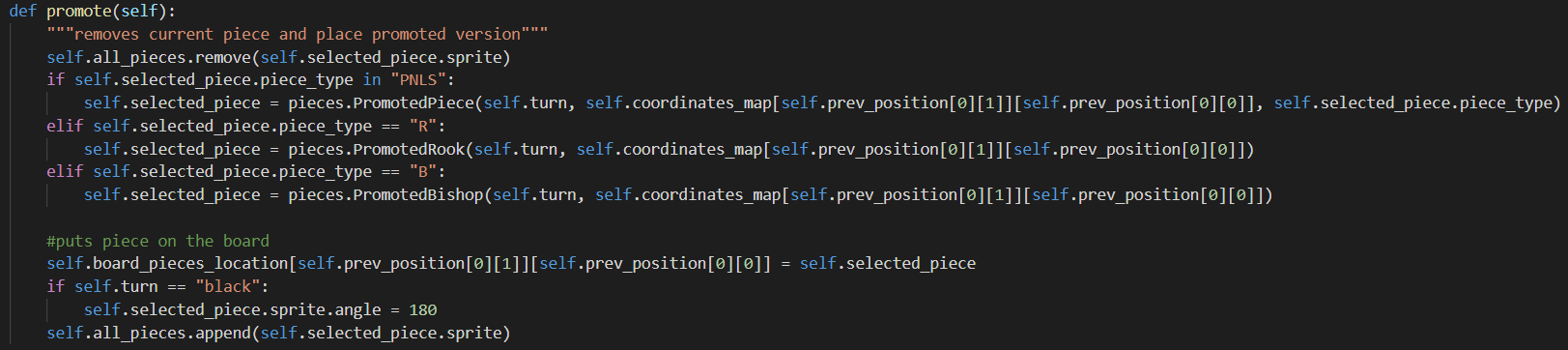


Image 25. Promotion

Depending on the piece type, the piece will promote into different promoted pieces. Refer to the second section to see how they move. Promotion works by removing the current piece from existence and then replacing it with an instance of the promoted piece’s class. The new piece will be placed on the same location on both the visible and invisible board.

**7. Pinning and checking**

In both chess and shogi, a piece is pinned if moving it causes the king to get checked. Likewise, a piece is pinning another piece if their trajectory towards a king is blocked by another piece. In chess, the bishop, rook and queen can pin pieces. In shogi, the bishop, rook and lance can pin pieces.

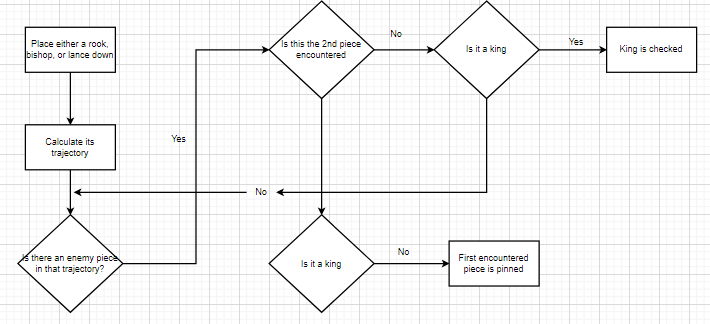


Image 25. Pinning logic

The above diagram shows the logic and steps used to determine whether a piece is pinning another piece.

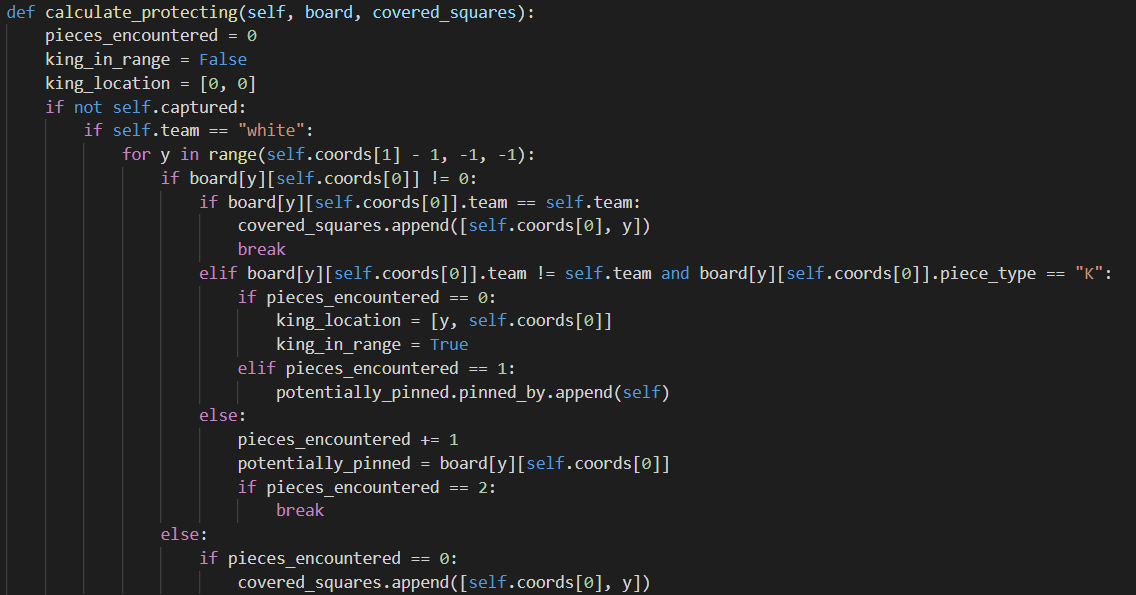


Image 25. Lance pinning and checking calculations

The code above is the calculations done for a lance piece. It works in exactly the same logic as the diagram shown before it. When a piece is pinned, its pinned\_by attribute is appended using this piece. In this code, an array is used instead of a Boolean for the sake of efficiency.

If the length of the array is more than 0, then the piece is clearly pinned. If a Boolean were to be used instead, calculating useful moves (covered in the next section) would prove to be more difficult as a Boolean does not store the kind of piece that is pinning it.

This method is available in every piece and is run before every switch in turns after a move. This is because this method serves a different purpose of inserting every possible move into the covered\_squares list. The covered\_squares list is used to prevent the king from moving into squares that are protected by opposing pieces.

Another purpose of this method is to calculate checks. This was shown in the diagram earlier. When the player’s piece is checking the king, it is placed into the checking\_pieces array in the main program file. The method returns True if opposing king is placed in check. The code for this paragraph will be shown in the next section.

**8. Useful moves and checkmating**

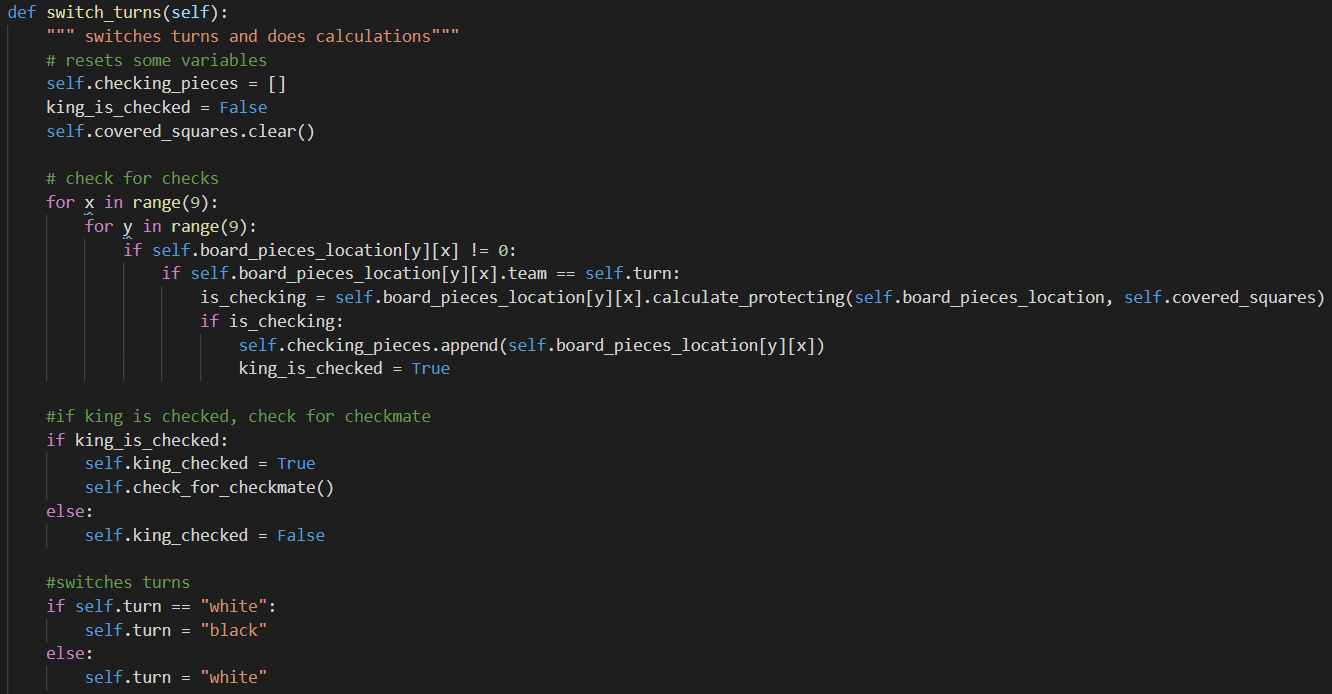


Image 26. Switch\_turns method

At the end of the player’s turn after they have made a move, the method switch\_turns is called. Like its name suggests, it switches turns from one to the other. However, it also does a lot of calculation that comes after making a move. As mentioned in the previous section, this method checks through every future possible move of every piece. When the calculate\_protecting method finds that a piece is checking the opposing king, the king\_checked attribute of MyGameWindow is set to True and the piece will be appended in the checking\_pieces list before running the check\_for\_checkmate method. These two variables and the covered\_squares variable are reset before every switch in turns.

Before explaining the check\_for\_checkmate method, the concept of a useful move must be understood. The term “useful moves” has been mentioned before earlier, and this refers to moves that actively block or remove a check on the player’s king. Although not an explicit variable in the program’s code, it helped conceptualize the solution to figuring out how to calculate legal moves when the king is in check.

The rules in which a piece is moving usefully, given that the king is in check, can be summarized into:  
1. If the move does not expose the king (if the piece is pinned by a long-range piece)  
2. If the move removes the checking piece  
3. If the move blocks the checking piece (if the checking piece is a long-ranged piece)  
4. If the move places the king at a safe square

These useful moves are the only possible moves the player can make when their king is in check. Although it could have just as easily been called “legal moves” instead, the word “useful” is used as the program filters through every possible move every piece could make if their king is **not** in check, and chooses only moves that are useful (blocks or removes the check)

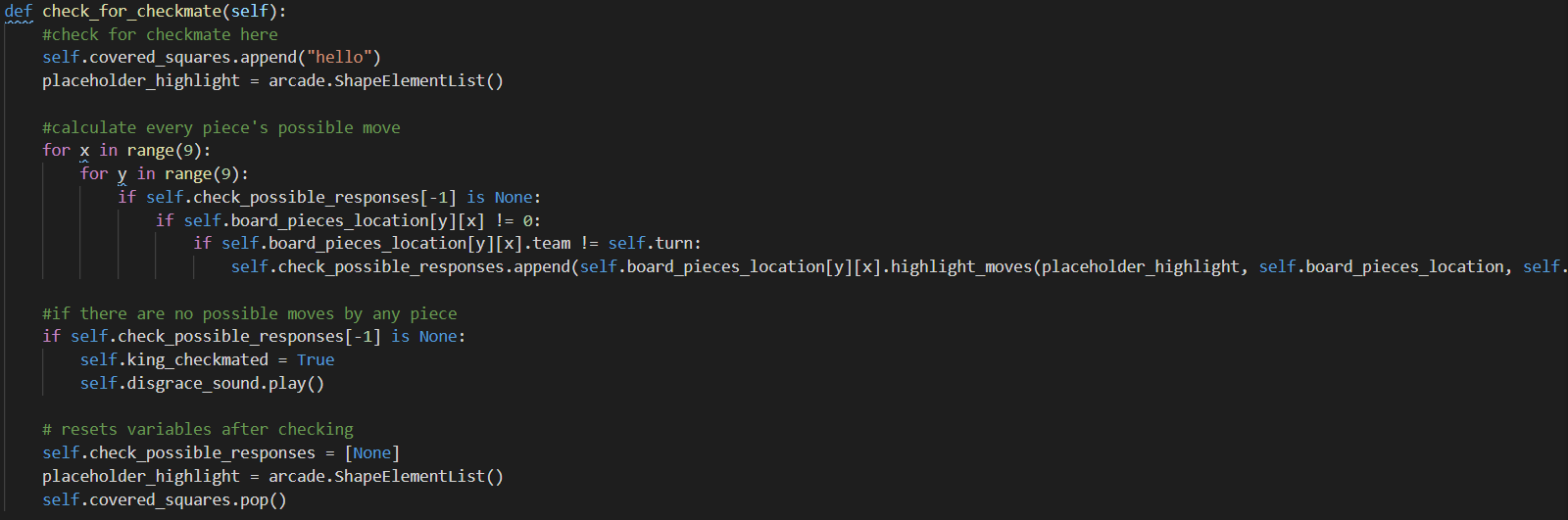


Image 27. Check\_for\_checkmate method

The check\_for\_checkmate method searches every possible moves of the pieces in the checked king’s team and appends only its useful moves into the check\_possible\_responses attribute. When a piece has no useful move, None is appended to the list. When even a single useful move is appended, the program stops calculating. This is because checkmate only happens when the opposing team does **not** have any useful moves (The king has nowhere to go, and no piece can remove the threat).

The highlight\_moves method is used because it calculates and highlights useful moves when the king is in check. However, this time it is used in a different way than usual as in this case, it is utilized solely for the sake of calculating whether a piece has any useful move. Hence, a decoy placeholder\_highlight variable is used as to not show highlighted squares when the program is calculating.

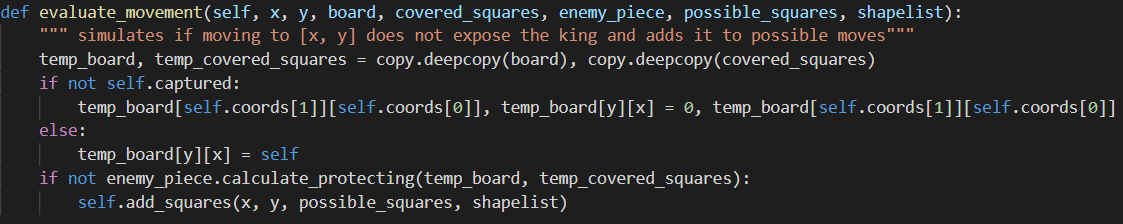


Image 28. Evaluate\_movement method

Inside the highlight\_moves method of pieces is another method found in the Pieces parent class called evaluate\_movement. When the king is in check or if the piece is pinned, this will run for every square in a piece’s normal movement. It works by creating a copy of the invisible board and simulating the move to that square. Since pinning can only be done by one piece and checking does not allow any movement other than the king if there are multiple checking pieces, only one enemy\_piece is input into the method as an argument.

After simulating the move, it will check the enemy\_piece’s possible movement and see if it threatens the king. If it does, then the square this method is calculating for is deemed not useful and will not be added to the possible\_squares list.

When there are no possible moves on the opposing team, that player loses as it is checkmate. When checkmate happens, an alert will be drawn on the screen to inform players the winner and the audio of a comical impression of a disappointed Asian parent will play (intended for the loser, obviously). To replay the game, simply close and re-open the program.

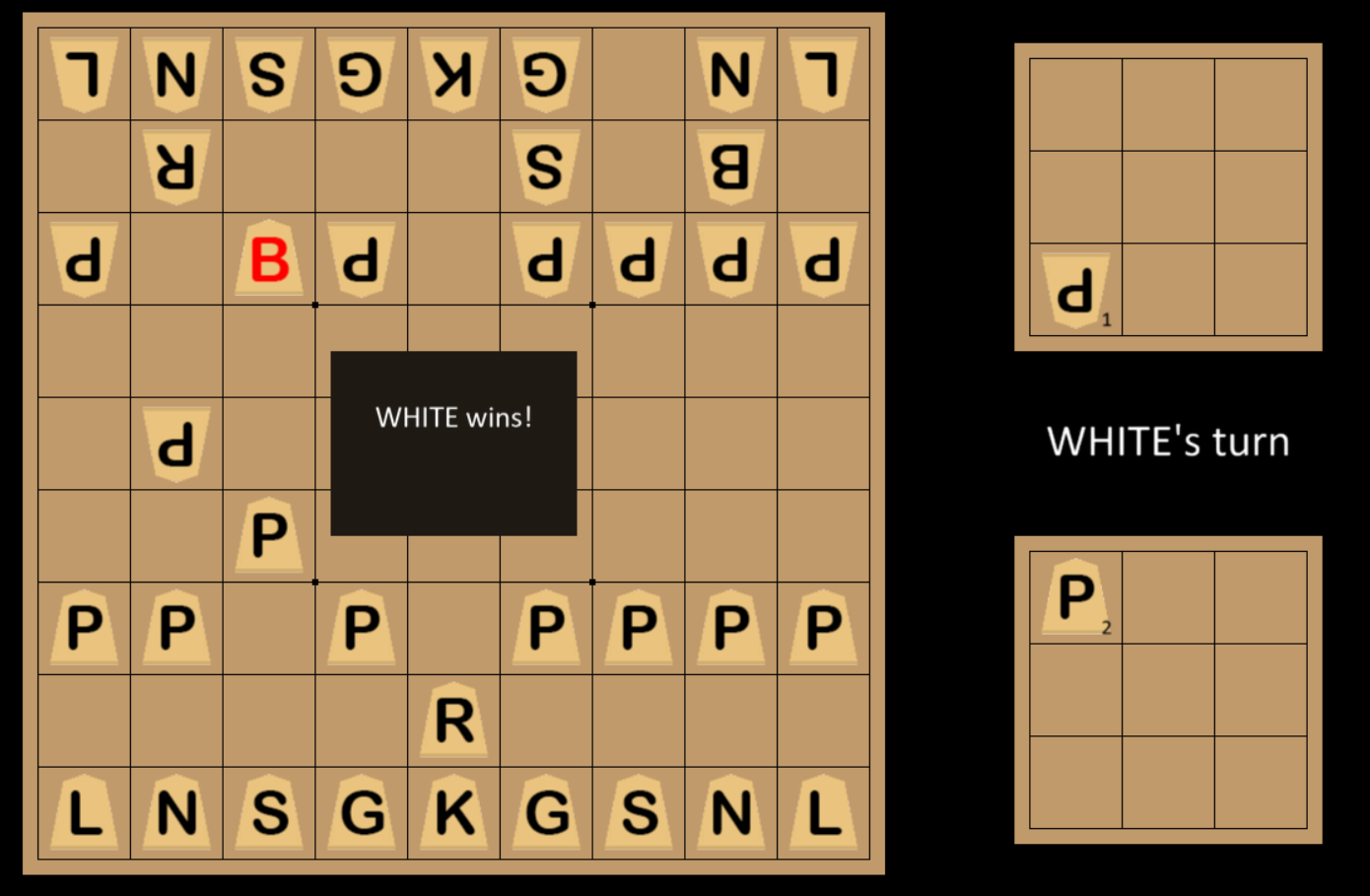


Image 29. Checkmate

**9. Game notation and records**

As of version 1.2 (after the presentation), the ability to see the game’s history and saves has been implemented. The following changes to the MyGameWindow class were made to make the update possible:

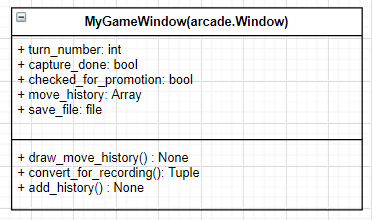


Image 30. New attributes and methods

To explain what each of these attributes do:  
- turn\_number is used to count the number of turns that have taken place so far. It is then written in the move history square on the right side on the screen (see image 31)  
- capture\_done is used to record whether a capture has been done in the current turn. This is used there is a different symbol used in the notation when a capture is done  
- checked\_for\_promotion is much like the previous attribute, except it records whether the game has prompted for promotion.  
- move\_history is used to help with both recording and drawing the move history on the right side of the screen. Whenever a new move is made, it is appended to this list.  
- save\_file is the save file used to store the records of the game. It is readable and every move is separated with a new line

To explain what each of the new methods do:  
- draw\_move\_history is the method used to draw and update the move history text on the right side of the screen  
- convert\_for\_recording converts the coordinates of the previous square and the destination square into notation form, hence the name. It returns a tuple consisting of 2 strings (the previous square and destination square)  
- add\_history combines the two values obtained from the previous method and uses it to create the correct notation used to represent the latest move. It then inserts it to the move\_history list as well as to save the move into the save\_file file.

Here is what the new screen will look like after the addition:

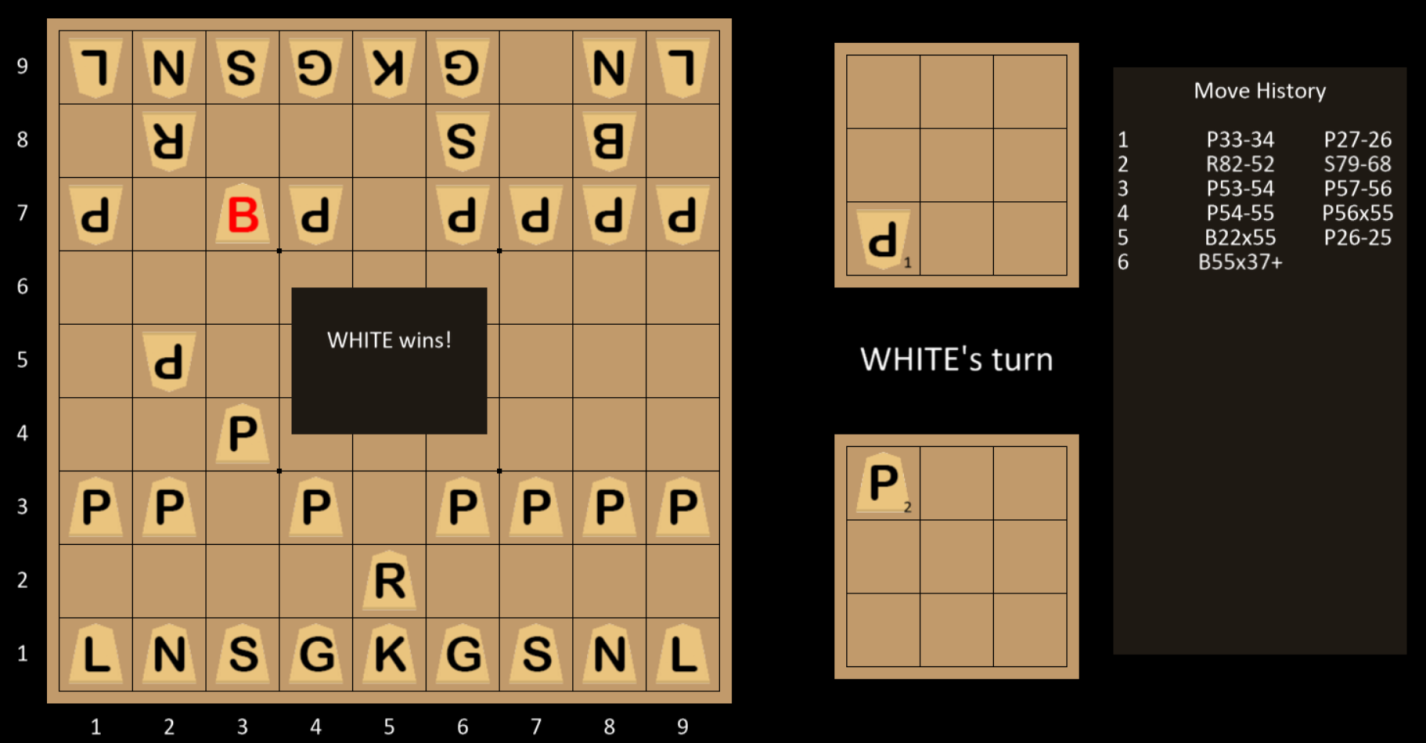


Image 31. New screen

To first explain, this program uses the western notation instead of the original Japanese notation as the Japanese notation makes use of kanji characters, of which not everyone is able to read. The western notation is formulated like so:

piece (origin) movement destination (promotion)

The first letter indicates the piece type, which is conveniently already a part of the game. They can be identified easily by noticing the letters on the pieces with the exception of promoted pieces, which is identified with an additional “+” sign before the letter. The origin is optional, as dropped pieces do not have an origin, but otherwise its origin is the piece’s original square before moving.

The movement type of a piece can be determined as one of three possible choices:  
1. “-“ – symbolizes normal movement towards the destination square  
2. “x” – symbolizes a capture on the destination square  
3. “\*” – symbolizes dropping that piece on the destination square

As the name suggests, the destination denotes the piece’s final square before switching turns. Finally, the optional promotion section of the notation is only used when a prompt comes up. A “+” is written if the player chooses to promote the piece while a “=” is written if the player declines the promotion. If an automatic promotion occurs, then the “+” is added to the end.

The logic done to implement this is simple enough that it hardly needs a diagram. Whenever a player makes a move, display the move notation on the right side of the screen. If the number of moves exceeds 20, the first line is erased and the entire list is moved upwards. Unfortunately, players are still unable to scroll up to see past moves. However, to compensate, the moves are automatically saved as a .txt file in a new subfolder named “saves”. In order to differentiate one save file from the next, it is named after the current date and time the game started.