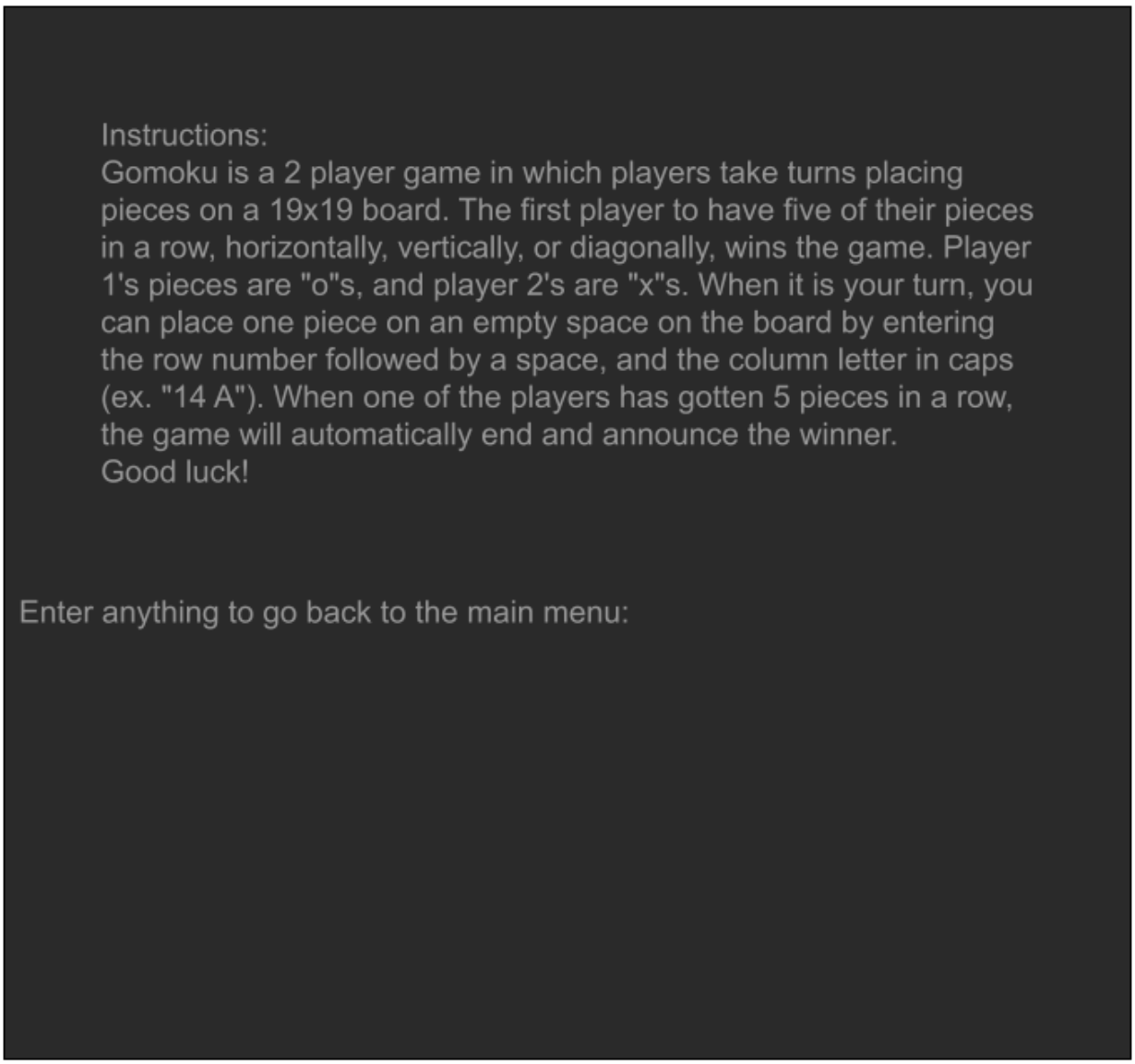
Gomoku Design Document

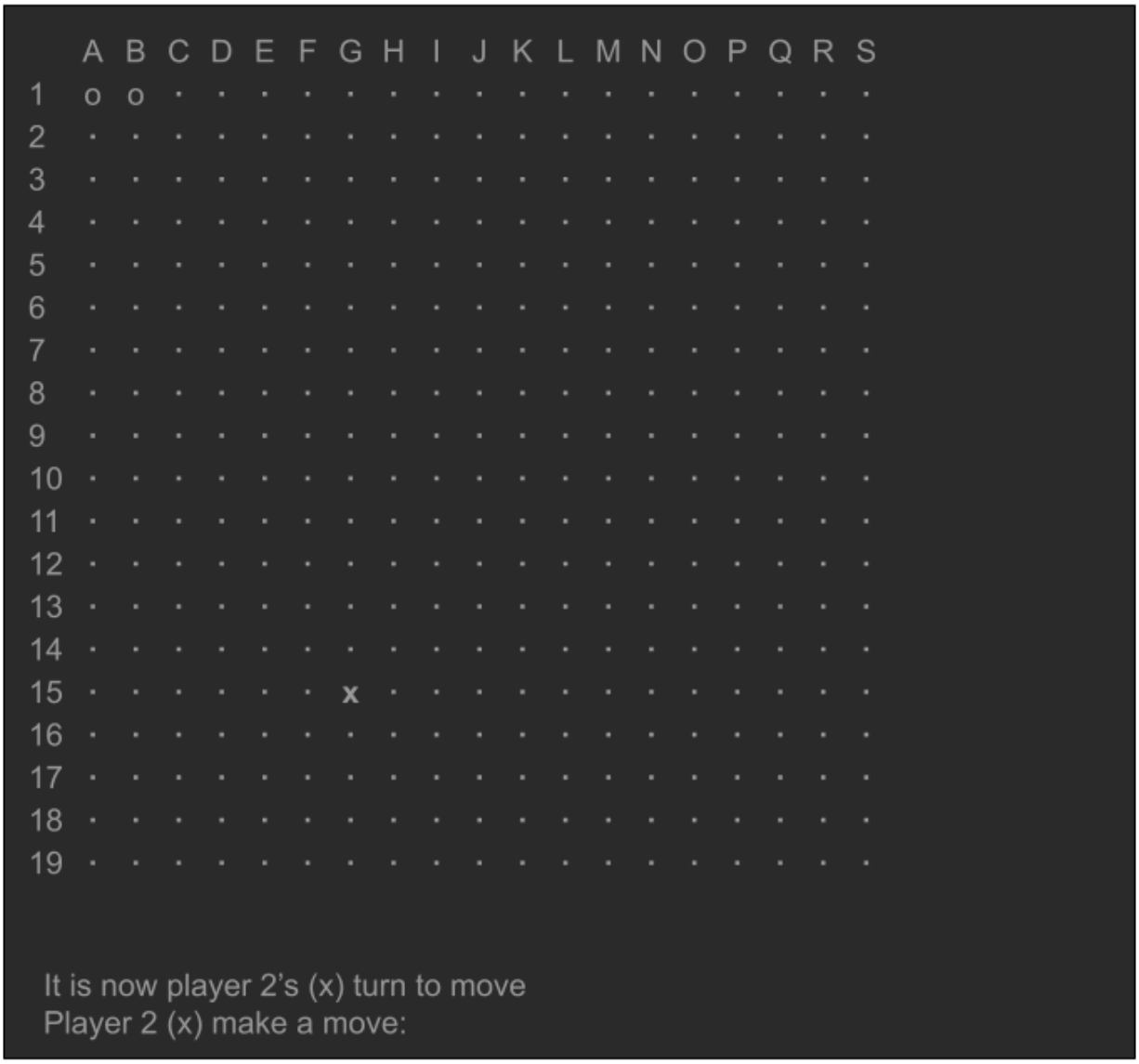
UI Design

Welcome screen and main menu

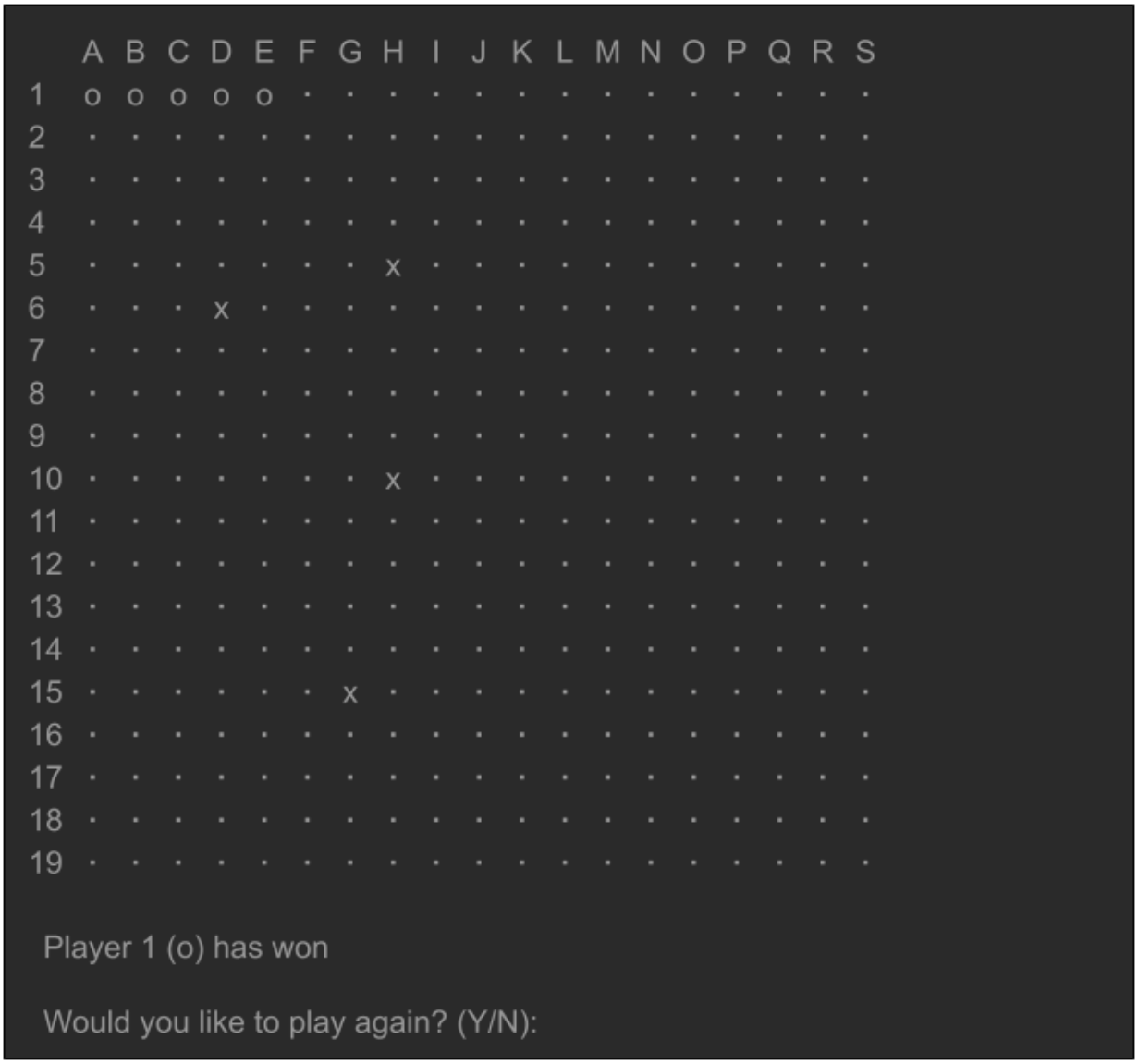
Instructions screen



Gameplay


A B C D E F G H I J K L M N O P Q R S
1 o o
15 x
It is now player 2's (x) turn to move
Player 2 (x) make a move:

Win screen



Implementation Design

Feature	**What does this feature do?**	**How will you implement this feature?**
Action Functions		
Gomoku board generator function	Generates a 19x19 board with rows labeled 1-19 and	- Create an empty list called board

`make_board()`	columns labeled A-S.	- Create a list for the number labels - Create a list for the letter labels and append the board list with the letter list as an item - Create the remaining row lists using a for loop (for i in range(0,20)). Create the row list with the first item being the label of the row by indexing the number label list using the for loop counter (i). - Use a sub for loop to add 19 dots as board spaces. - Once the sub for loop completes, append the board list with the completed row list - Return board
Gomoku board print function `print_board(my_board)`	Prints the gomoku board 2D list with even spacing. An extra space is added after the number labels from row 0-9 in order to make up for the extra space taken by double digit labels	- Use a for loop that repeats 19 times to print each row - If the row label is a number form 1-9, that character is printed with 2 spaces instead of 1 to compensate for the double digit labels. The remaining items in the board row are printed with 1 space - Else prints each board row list item with a space
Coordinate format converter (helper function) `convert_to_coordinates(user_input)`	Convert user coordinate input from number-letter format to number-number format	- Call .split() on argument user_input (number-letter format) and save in variable split - Convert split[0] (number) to int using int()

		- Convert split[1] (letter) to int by using ord()-64 - Save numerical coordinates in a tuple coordinates - Return coordinates
Move x piece to input coordinates `move_x(user_input)`	Changes item in the 2D gomoku board list at specified indexes (coordinates in number-letter format) to “x”	- Call `convert_to_coordinates(user_input)` function with user_input as argument and store returned coordinate tuple in variable point - Store point[0] in variable x - Store point[1] in variable y - Change the value of the item at gomoku_board[x][y] to “x”
Move o piece to input coordinates `move_o(user_input)`	Changes item in the 2D gomoku board list at specified indexes (coordinates in number-letter format) to “o”	- Call `convert_to_coordinates(user_input)` function with user_input as argument and store returned coordinate tuple in variable point - Store point[0] in variable x - Store point[1] in variable y - Change the value of the item at gomoku_board[x][y] to “o”
Check functions		
Coordinate input validity checker function (for number-letter format) `input_valid(user_input)`	Check if user input is valid and coordinates are on the board	- If length of user input is not 3 or 4, return False - If length of user input is 3, and the first character is not a

		number from 1-9, return False - If length of user input is 4 and first two characters are not a number from 10-19, return False - If the last character is not a capital letter from A to S, return False - If none of the above conditions are met, return True
Coordinate input validity checker function (for number-number format) `on_board(x, y)`	Check if coordinates are on the board	- If first argument x is a number from 1-19 and second argument y is also a number from 1-19, return True - Otherwise return False
Move validity checker function `move_valid(user_input)`	Checks if user input is a valid move. Checks if input is on the board and if space is unoccupied.	- Call `convert_to_coordinates(user_input)` function and store returned coordinate tuple in variable point - Set x = point[0] and y = point[1] - If `on_board(x, y)` returns True and the coordinate on the board gomoku_board[x][y] is an empty space `"."`, return True - Otherwise return False
Win condition checks		
Horizontal win condition checker function `has_horizontal_win(user_input, player)`	Determines if after the current player's last move, there are 5 of the current players pieces in a horizontal row	- Create function with two parameters `has_horizontal_win(user_input, player)` - Convert `user_input` from number-letter

		format to number-number format by calling `convert_to_coordinates(user_input)` function and store returned tuple in variable `point` - User `player` to determine the piece to check for. If player is 1, piece is `"o"`, if player is 2, piece is `"x"` - Create variable `x` and give it value of `point[0]` - Create variable `y` and give it value of `point[1]` - Create list of pieces that are in a row `pieces_in_row` and add tuple `point` as an item - Use a for loop that repeats twice to determine `xshift`. (1st rep, x-shift = 1, 2nd rep, x-shift = -1) This will be the number added to the x coordinate to check the next horizontal piece. This will allow both directions from the original move to be checked - Use a for loop that repeats 5 times to check spaces horizontally adjacent to `point` by adding `xshift` to the previous `x` variable - If space contains the current player's piece, add the x and y coordinates to the list of `pieces_in_row` as

		a tuple. - If the board space does not contain the current player's piece, break and check in other direction - If the length of the `pieces_in_row` list is equal to 5 or greater, break loops and return True
Vertical win condition checker function `has_vertical_win(user_input, player)`	Determines if after the current player's last move, there are 5 of the current players pieces in a vertical row	- Create function with two parameters `has_vertical_win(user_input, player)` - Convert `user_input` from number-letter format to number-number format by calling `convert_to_coordinates(user_input)` function and store returned tuple in variable `point` - User `player` to determine the piece to check for. If player is 1, piece is `"o"`, if player is 2, piece is `"x"` - Create variable `x` and give it value of `point[0]` - Create variable `y` and give it value of `point[1]` - Create list of pieces that are in a row `pieces_in_row` and add tuple `point` as an item - Use a for loop that repeats twice to determine `yshift`. (1st rep, y-shift = 1, 2nd rep, y-shift = -1) This will be the number added to the

		y coordinate to check the next vertical piece. This will allow both directions from the original move to be checked - Use a for loop that repeats 5 times to check spaces vertically adjacent to `point` by adding `yshift` to the previous `y` variable - If space contains the current player's piece, add the x and y coordinates to the list of `pieces_in_row` as a tuple. - If the board space does not contain the current player's piece, break and check in other direction - If the length of the `pieces_in_row` list is equal to 5 or greater, break loops and return True
45° diagonal win condition checker function `has_45diagonal_win(user_input, player)`	Determines if after the current player's last move, there are 5 of the current players pieces in a 45° diagonal row	- Create function with two parameters `has_45diagonal_win(user_input, player)` - Convert `user_input` from number-letter format to number-number format by calling `convert_to_coordinates(user_input)` function and store returned tuple in variable `point` - User `player` to determine the piece to check for. If player is 1, piece is `"o"`, if player is 2, piece is

		`"x"` - Create variable `x` and give it value of `point[0]` - Create variable `y` and give it value of `point[1]` - Create list of pieces that are in a row `pieces_in_row` and add tuple `point` as an item - Use a for loop that repeats twice to determine `shift`. (1st rep, shift = 1, 2nd rep, shift = -1) This will be the number added to the x and y coordinates to check the next horizontal piece. This will allow both directions from the original move to be checked - Use a for loop that repeats 5 times to check spaces horizontally adjacent to `point` by adding `shift` to the previous `x` and `y` variables - If space contains the current player's piece, add the x and y coordinates to the list of `pieces_in_row` as a tuple. - If the board space does not contain the current player's piece, break and check in other direction - If the length of the `pieces_in_row` list is equal to 5 or greater, break loops and return True
315° diagonal win condition	Determines if after the current	- Create function with

checker function `has_315diagonal_win(user_input, player)`	player's last move, there are 5 of the current players pieces in a 315° horizontal row	two parameters `has_315diagonal_win(user_input, player)` - Convert `user_input` from number-letter format to number-number format by calling `convert_to_coordinates(user_input)` function and store returned tuple in variable `point` - User `player` to determine the piece to check for. If player is 1, piece is `"o"`, if player is 2, piece is `"x"` - Create variable `x` and give it value of `point[0]` - Create variable `y` and give it value of `point[1]` - Create list of pieces that are in a row `pieces_in_row` and add tuple `point` as an item - Use a for loop that repeats twice to determine `xshift` and `yshift`. (1st rep, `yshift = 1` and `xshift = -1`, 2nd rep, `yshift = -1` and `xshift = 1`) These will be the numbers added to the x and y coordinates to check the next horizontal piece. This will allow both directions from the original move to be checked - Use a for loop that repeats 5 times to check spaces

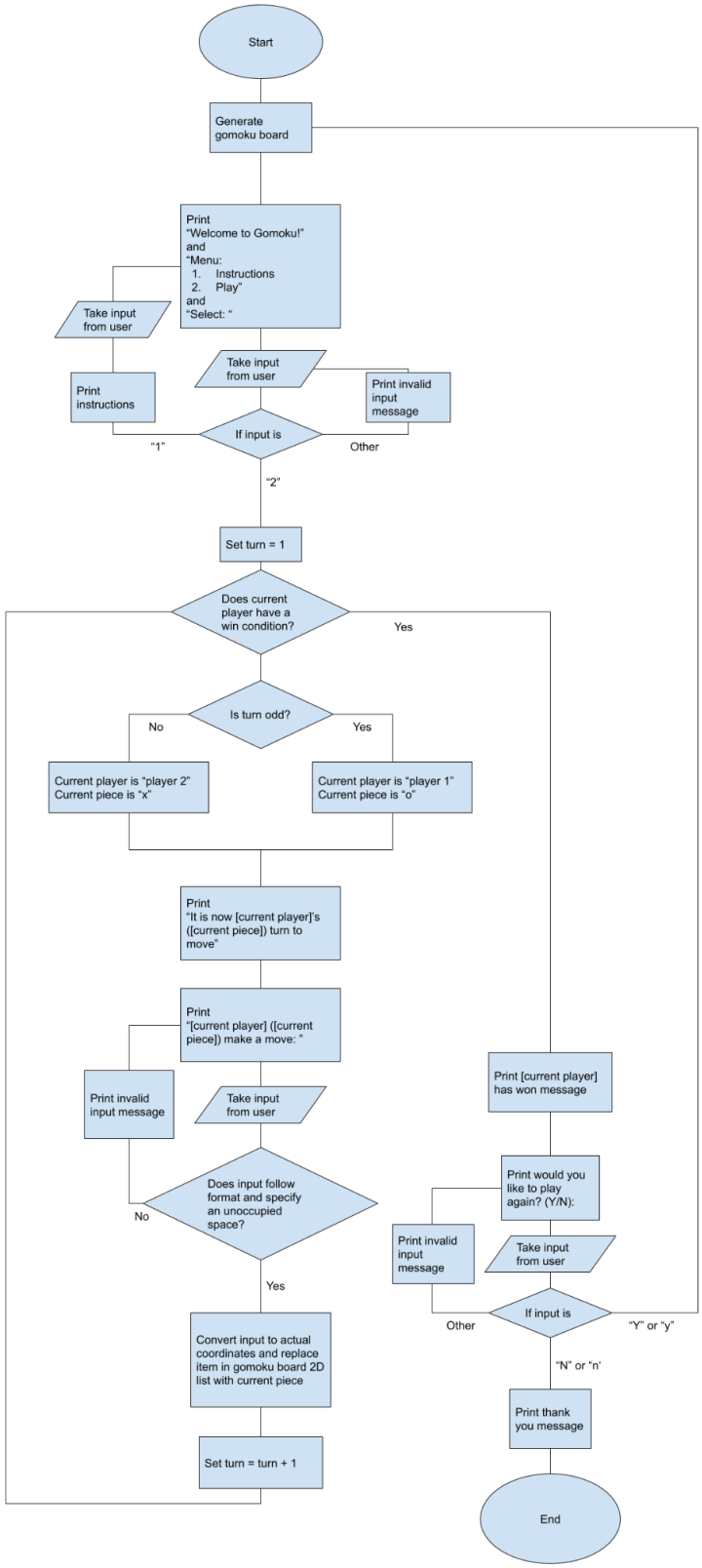
		horizontally adjacent to `point` by adding `xshift` and `yshift` to the previous `x` and `y` variables - If space contains the current player's piece, add the x and y coordinates to the list of `pieces_in_row` as a tuple. - If the board space does not contain the current player's piece, break and check in other direction - If the length of the `pieces_in_row` list is equal to 5 or greater, break loops and return True
All win condition checker function `has_win(user_input, player)`	Checks for overall win using the previous functions	- Calls all directional win condition check functions - If any of then return True, return True - Otherwise return False
Game code		
Play again option	Allows the user to choose whether to quit or play again	- The game code is enclosed in a grand while loop - When the game ends, the user is asked if they want to play again - User input is taken using input() - Using if statement, If user inputs "N" or "n", a thank you message is displayed, the loop is broken, and the program ends. - Using elif statement If user inputs "Y" or "y", no action is taken,

		and the while loop restarts the game - If user input is not valid, an error message is displayed and user is reprompted for an input by enclosing the input prompt and if/elif./else statements in a loop and using else statement.
Main menu	Menu options for instructions and starting the game are displayed, and the user is prompted to select one of them by inputting the option number.	- Use print() to display menu options - Use a while loop and input() to prompt the user to make a selection - Checking with if statement, If user input is not a menu option, display an error message and let while loop reprompt user for input - Checking with if statement, if user input is a menu option, store input in a variable called “selection” and break the while loop
Instructions page	If menu option 1 is selected, the instructions are printed, along with an input prompt to enter any input to return to main menu.	- Use an if statement to check that user input (variable “selection”) is “1” - Use print() to print instructions for the game - Use input() and display a prompt to allow the user to return to main menu - If user makes any input, take no action, and allow the grand loop to return the user to main menu

Start game	If menu option 2 is selected, the game begins.	- Use an if statement to check that user input (variable “selection”) is “2” - Continue to gameplay code
Check turn	The turn number is stored in a variable. When the game begins, the variable is set as turn = 1. At the end of each turn, after a move is made, turn is increased by one. The player whose turn it is is determined by whether the turn number is odd or even, thus alternating the turns.	- Before the game starts (before while loop begins), set variable turn = 1 - Use % operator and if statement to determine if turn is even or odd, and store player number and piece in a tuple - If turn is odd, store 1 and “o” in a tuple player = (1, “o”) - If turn is even, store 2 and “x” in a tuple player = (2, “x”) - At the end of each turn (end of while loop), after a move has been made, add 1 to the turn variable
Display current board	Print the board with all moves made so far.	- Call the print_board(my_board) function with current edited board as the argument
Check win and end game	Checks if the current player has a win after making their move	- Call the `has_win(user_input, player)` which checks for win condition in every direction - If the function returns true, the turn loop is broken and the play again option is given
Take input, check, and make move	Prompts the current player to make a move and either repromts if move is invalid, or makes the move if move is	- Within a while loop, prompt current player (player[0]) to make a move

	valid.	- Call the `input_valid(user_input)` function to check for formatting - Print error message and repromt if formatting is invalid - Call `move_valid(user_input)` function to check if move is on an unoccupied piece on the board - If move is invalid, print error message and repromt - If input is good, call `move_o(user_input)` if player[0] is 1 or `move_x(user_input)` if player[0] is 2

Implementation Detail



Link to drawing:
https://docs.google.com/drawings/d/10VC6eqsPU4UJWoPAf40LlAf7I9i4a_lPsYBrePsQAUM/edit?usp=sharing