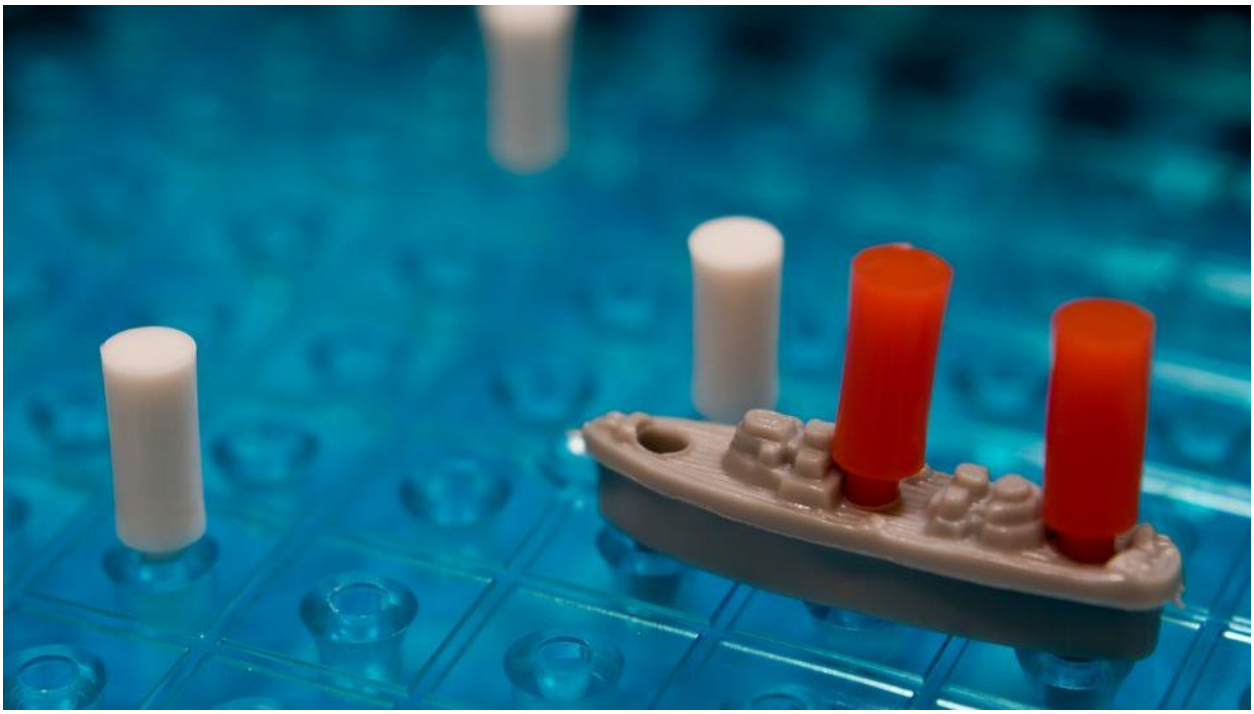
15-110 Hw6 - Battleship

Hw6 and its check-ins are organized differently from the other assignments. If you haven't already done so, you should read the **Hw6 General Guide** to understand how this assignment works.

Project Description



Hi there! So you've decided to do Battleship. Well you're in for a treat, because by the end of only 3 weeks, you will have a fully working, simplified version of this classic game. Let's get into this!

If you've never played Battleship before, try an example game here: https://www.mathsisfun.com/games/battleship.html

Click on the following links to read the instructions for each week's assignment:

Hw6 Check-in 1 - due Monday 11/18

Hw6 Check-in 2 - due Monday 11/25

Hw6 - due Wednesday 12/04

Hw6 Check-in 1 - due Monday 11/18

This week we will begin to create the framework behind the game by making functions that will create an empty grid, create ships, add them to the grid, and finally draw the grid. By the end of the stage, you will have code that automatically produces a starting grid for the computer

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Generate an Empty Grid **[5pts]**

Write a function emptyGrid(rows,cols) which creates a new 2D list (called a grid) with rows number of rows and cols number of columns. The value of each grid[row][col] should be 1, which stands for an empty spot that has not been clicked. Return the new 2D list.

Note that we'll use a number system to represent all cells that can show up in the grid. This number system has been provided as global variables at the top of the file. Specifically:

EMPTY_UNCLICKED = 1
SHIP_UNCLICKED = 2
EMPTY_CLICKED = 3
SHIP_CLICKED = 4

To test this function, run testEmptyGrid().

Step 2: Create Ships **[10pts]**

Write a function createShip() which chooses a random row in the range [1,8] and random column in the same range to be the center point of a ship. We choose 1-8, so that it cannot put a ship in the last row or column which wouldn't fit on the board.

After choosing a row and column as the center, your code should choose a random number 0 or 1 to decide if the ship will be vertical or horizontal. If it chose vertical, create a ship within the same column, where the ends are one above and one below the

chosen center - e.g., row-1, row, row+1. Similarly, if it chose horizontal, create a ship in the same row, where the ends are one column left and right of center - e.g., col-1, col, col+1.

Each ship will be a list of 3 coordinates (two-element lists). Return the ship (2D list) created.

To test this function, run testCreateShip().

Step 3: Validate Ships **[10pts]**

We'll need to check if randomly-generated ships can actually be added to the grid. Write a function checkShip(grid, ship) that iterates through the given ship and checks if each coordinate in the ship is 'clear'. A coordinate is clear if the corresponding location on the given grid is empty (EMPTY_UNCLICKED). It should return the list of not-clear coordinates, or the empty list if all coordinates are clear.

To test this function, run testCheckShip().

Step 4: Add Ships to Board **[15pts]**

Write a function addShips(grid, numShips) which loops until it has added numShips ships to the grid. For each time through the loop, it should create a ship using createShip(), then checkShip for that ship on the given grid. If the returned not-clear list is empty, then the ship can be placed.

To place the ship, iterate through each coordinate of the ship, and set the grid at that coordinate to 2 (SHIP_UNCLICKED), then add 1 to the current count of ships. The function should return the grid that was modified to add numShips ships.

To test this function, run testAddShips(). Note that this function involves randomness, so you will need to check if the grid is actually modified yourself.

Step 5: Store Initial Data **[5pts]**

Now that we can make the grid, we need to set up the simulation itself!

You'll need to update makeModel to set up several variables. These variables will be a part of data, so make sure to define them as data["name"] = value.

First, store data about the board dimensions. You should store the number of rows, number of cols, board size, and cell size. You can start with 10 rows, 10 cols, and a 500px board size. You can then compute the cell size based on the board size and number of rows/cols.

Next, store data about the board. You'll need to keep track of two boards- one for the computer, and one for the user. You should store the number of ships on the board (start with 5), then set both the computer board and the user board to be a new empty grid, by calling your emptyGrid() function. Finally, update your computer board by calling your addShips() function.

Step 6: Draw the Grid **[10pts]**

Now we just need to add graphics. Write drawGrid, which draws a grid of rows x cols squares on the given canvas. Each square should have the cell size you determined in the previous step. If the cell in the given grid at a coordinate is SHIP_UNCLICKED, the square should be filled yellow; otherwise, it should be filled blue. Ignore the showShips parameter for now; it will be used in a future step.

Hint: you did something very similar to this in Hw3 Check-in 1...

Finally, update makeView so that it calls drawGrid on the computer's board and canvas. You can use False for the showShips variable for now. Once this is done, when you run the simulation, you should see the computer's starter board in the computer window!

Hw6 Check-in 2 - due Monday 11/25

In the second stage of the project, you will write code that lets the user select where to place ships on their grid. You will do this by detecting where the user is clicking, and whether their clicked cells form a legal ship or not.

Step 0: Written Assignment **[45pts]**

In addition to completing the steps described below, there is a short written assignment on the week's material. You can find the written assignment on Gradescope, and linked on the Assignments page of the course website.

Step 1: Check Direction **[10pts]**

First, write two functions to check whether a set of three coordinates are laid in a specific direction (vertical or horizontal). Write isVertical, which takes in a ship (recall from last week that a ship is a 2D list of coordinates) and returns True if the ship is placed vertically, or False otherwise. Recall that a ship is vertical if its coordinates all share the same column, and are each 1 row away from the next part.

Then write isHorizontal, which takes in a ship and returns True if the ship is placed horizontally, and False otherwise. This should work in a similar manner to isVertical, except that the dimensions are flipped.

To test your functions, run testIsVertical() and testIsHorizontal(). Make sure to uncomment the week2Tests() call in order to run these!

Step 2: Detect Clicked Cells **[10pts]**

Next, we need to handle mouse events, to detect where a user has clicked on the board. Write the function getClickedCell(data, event) which takes the simulation's data dictionary and a mouse event, and returns a two-element list holding the row and col of the cell that was clicked.

Recall that the event value holds event.x and event.y; you need to convert these to the row and col. How can you do this? You have two choices- either derive the row and col mathematically, or iterate over every possible row and col in the board, calculate each (row, col) cell's left, top, right, and bottom bounds, and check if the (x,y) coordinate falls within those.

This step is difficult to test on its own; you'll test it in the simulation in a few steps instead.

Step 3: Handle User Clicks **[20pts]**

Now we can write a function that will actually handle user clicks and let the user add ships to the board. To do this, we need to keep track of a temporary ship that the user is adding cells to; add a variable for this to the data dictionary in makeModel.

Implement the function clickUserBoard(data, row, col), which handles a click event on a specific cell. First, check if the clicked location is already in the temporary ship list; if it is, return early. This will keep the user from adding multiple cells in the same location. Assuming the clicked cell is not in the temporary ship, add it to the temporary ship value.

If the temporary ship contains three cells, we'll try to add it to the board, since all of our ships will be three units long. However, ships should only be added if they do not overlap any already-placed ships, and if they cover three connected cells (either vertically or horizontally). You can test this by calling checkShip(), isVertical, and isHorizontal().

If the temporary ship is legal, add it to the user's board by updating the board at each cell to hold the value 2 (SHIP_UNCLICKED). If it is illegal, print an error message to the interpreter and reset the temporary ship to be empty.

Finally, we need to keep track of how many ships the user has added so far. Add one more variable to data in makeModel, in order to track the number of user ships; it should start as 0. Then, in clickUserBoard(), add one to that variable if a ship is added. At the end of the function, check if the user has added 5 ships, and tell them to start playing the game if so. And at the beginning of the function, exit immediately if 5 ships have already been added, to keep the user from adding too many ships.

Again, this step is difficult to test on its own, we'll test it soon.

Step 4: Update the Graphics **[10pts]**

We'll need to include a graphics function that draws the temporary ship. Write drawShip(data, canvas, ship), which draws white cells for each component of the given ship.

Then update the makeView() function at the top of the file. We need to draw the computer board, the user board, and the ship. Call drawGrid() on the user's board and canvas (setting showShips to True), then call drawShip() on the temporary ship in data. Note that the temporary ship should be drawn on the user's board.

Step 5: Manage Mouse Events **[5pts]**

Now we can finally put it all together by capturing and handling mouse events. In the mousePressed() function at the top of the file, use your getClickedCell() function to determine the row and col of the cell that was clicked. Next, note that we've added an additional parameter to the mousePressed() function. This parameter, board, is "user" if the click happened on the user's board, or "comp" if the click happened on the computer's board. If it is "user", call clickUserBoard() with the row and col, to update the temporary ship and board.

With this step, you can finally test your code so far! If you run into errors, try printing out what each helper function returns, to determine where the error is occurring across all your functions.

Hw6 - due Wednesday 12/04

In the final stage, you will implement the actual gameplay of Battleship. This will involve letting the user click on cells on the computer's grid, having the computer choose cells to click on the user's grid, and determining when the game is over.

Step 0-A: Complete Check-in 1 **[20pts]**

If you got a perfect score on Hw6 Check-in 1 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 1 and use it to update your Check-in 1 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 0-B: Complete Check-in 2 **[20pts]**

If you got a perfect score on Hw6 Check-in 2 (the project part), congratulations; this step is already done! Go to the next step.

Otherwise, go back to your Gradescope feedback on Check-in 2 and use it to update your Check-in 2 code. This is your chance to implement any features you might have missed before, and fix any code that isn't working.

Step 1: Handle User Guesses **[20pts]**

First, we need to update the simulation code to let the user guess where ships are on the computer's board. This will involve checking the spot that was clicked, updating it as appropriate, and drawing clicked cells on the grid.

First, write the function updateBoard(data, board, row, col, player), which updates the given board at (row, col) based on a player's click. If the user clicks on a cell with value SHIP_UNCLICKED (2), the board should update that cell to instead be SHIP_CLICKED (4). Otherwise, if the user clicks on a cell with value EMPTY_UNCLICKED (1), it should update to be EMPTY_CLICKED (3).

Next, write the function runGameTurn(data, row, col), which manages a single turn of the game after a user clicks on (row, col). First, check whether (row, col) has already been clicked on the computer's board (ie, if it is SHIP_CLICKED or EMPTY_CLICKED); the rest of the function should only run if this move is valid. Then call updateBoard() with

the appropriate parameters (including "user" as the player) to update the board at that spot.

Now we need to update the simulation code. In mousePressed, if the computer's board has been clicked and all the user's ships have been placed (ie, gameplay has started), call runGameTurn on the clicked cell.

Finally, update drawGrid to account for our two new types of cells. SHIP_CLICKED cells should be drawn as red, and EMPTY_CLICKED cells should be drawn as white. We'll also finally use our parameter showShips to make the game more interesting. Battleship is too easy if you can see your opponent's ships, so if showShips is False, draw SHIP_UNCLICKED cells as blue (to hide them). If you correctly set the drawGrid calls in makeView, this should draw the two boards properly (with the computer ships hidden and the user ships showing up).

To test your code, try clicking on cells in the computer's board. They should be changed to red or white!

Step 2: Handle Computer Guesses **[10pts]**

For every guess the user makes, we want the computer to make a guess as well. Write the function getComputerGuess(data). This function should return a cell that the computer will 'click' on the user's board. We'll have the computer select cells completely randomly; use the random.randint() function to pick the row and col. Finally, to make sure that the computer doesn't click the same cell twice, use a while loop to keep picking new (row, col) pairs until you find one that hasn't been clicked in the user board yet.

Now update the runGameTurn function to get the computer's guess (by calling getComputerGuess()), then run updateBoard() to update the user board at that location (with "comp" as the player). This should happen immediately after the user's guess is added to the computer's board.

To test your code, try running the simulation again, and clicking on a cell on the computer board. A cell should automatically be picked on the user's board as soon as you make your selection.

Step 3: Detecting a Winner **[15pts]**

Finally, we want to determine when the game ends, and who wins. We'll need to add a new variable to data in makeModel for this- something to keep track of the winner. It can start as None.

To do this, we'll write the function isGameOver(data, board), which checks whether the game is over for the given board. The game is done if there are no SHIP_UNCLICKED cells left in the board- in other words, when every ship has been clicked. Return True if the game is over for that board, and False otherwise.

The best place to check whether the game is over is right after the board is updated. In updateBoard(), call isGameOver on the given board. If the result is True, set the winner variable in data to the player parameter

Now update makeView() to draw a special message for the end of the game. If the winner is "user", print a congratulations message. If the winner is "comp", tell the user that they lost.

Finally, in mousePressed(), only let the user click on cells when a winner hasn't been chosen yet (when the data variable is None).

To test your code, try to win the game! You might have to lose on purpose to test the computer-winning scenario.

Step 4: Detecting a Draw **[10pts]**

It isn't very hard to beat a computer that makes guesses entirely randomly, so we'll add one extra feature to the game- declaring a draw. We'll say that if half the board (50 cells) is clicked with no winner, the result is a draw instead.

Add two data variables in makeModel- one that holds the max number of turns (50), and one that holds the current number of turns (which starts at 0). Then, in runGameTurn, once both the user and the computer have made their moves, add one to the number of turns made so far. In that same function, check whether the number of turns is equal to the max number of turns. If it is, set the winner variable in data to "draw".

Then add one final message in makeView()- if the winner is "draw", tell the user they're out of moves and have reached a draw.

Make sure to test the simulation to see what it does if you reach the draw state!

Step 5: Restarting the Game **[5pts]**

We'll add one last feature- letting the user play again. This will be done by detecting if the user presses the Enter key after the game is over.

Add a message in makeView after each of the possible end-game messages that tells the user to press Enter if they want to play again. Then, in keyPressed, check if the user pressed enter with event.keysym == "Return". If they did, reset the game.

The easiest way to reset the game is to reset all the data variables! You can do this very simply by calling makeModel(data) again.

Make sure to test this last feature by pressing Enter after you finish a game, to see if you can play a game. Once that's working, congratulations- you're done!