Student Names: Riccardo Risi & Roberto Forner (GROUP 20)

Python

Final Project

Introduction

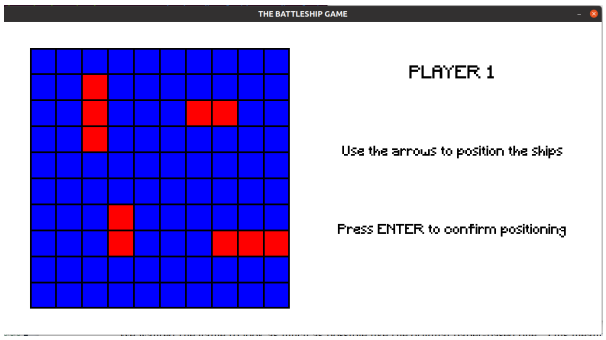
The aim of this project was to build the game of Battleship using the Python programming language. Battleship is a strategy-type board game that can be played by two people: each player has a ruled grid where he can position his fleet of warships, the exact location of these ships remaining unknown to the other player. Taking turns, each player shoots at the grid of his opponent by declaring a pair of coordinates (row and column of the grid). If the player announces a pair of coordinates where a ship is located, the other player has to declare that one of his ships was "hit", or else that his ships were "missed". The game ends once a player loses all his ships.

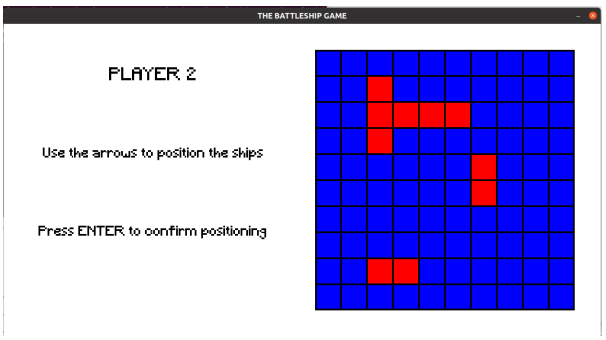
Specifications

As previously stated, our goal was to implement the Battleship game using Python. To begin with, we decided that we would have used the Pygame module, which is a set of Python modules designed for writing video games. Indeed this module would save us from the burden of writing functions for handling the screen visualization of the game, as well as for handling user inputs. This way we could focus on developing the game logic only, which is the core of this project.

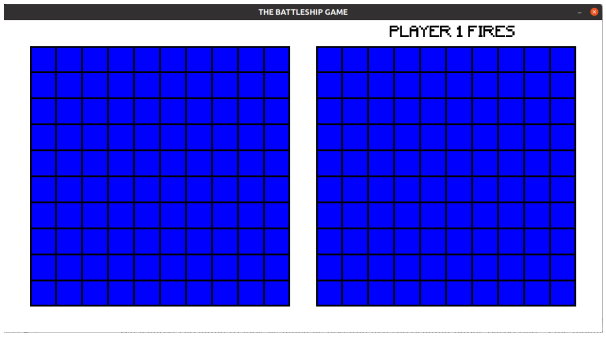
We wanted the game to look as much as possible like the original paper-based one. This meant, firstly, that each player should have had a grid to position his ships, thus, two 12x12 grids had to be drawn on the screen. Pygame came in handy for this purpose, allowing us to draw grids with just a few lines of code.

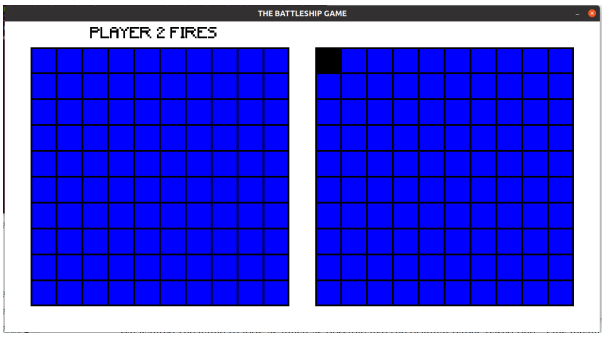
At the beginning of the game, each player has to position his ships on the board. In the beginning, the program asks Player-1 to place his ships on the board (while Player-2 is looking away from the screen) and, once he is done, his ships will disappear from the screen and Player-2 will be prompted to position his ships. Each player may position a total of four ships: two 3x1 and two 2x1. Two vertically and two horizontally. The ships pop-up on the screen one by one and the player is prompted to use the arrows to move them on the board as he likes. Once he presses enter, the position is locked and the ship cannot be moved anymore. Furthermore, the program does not allow the user to move a ship outside of the board, and not even position a ship on top of another one.



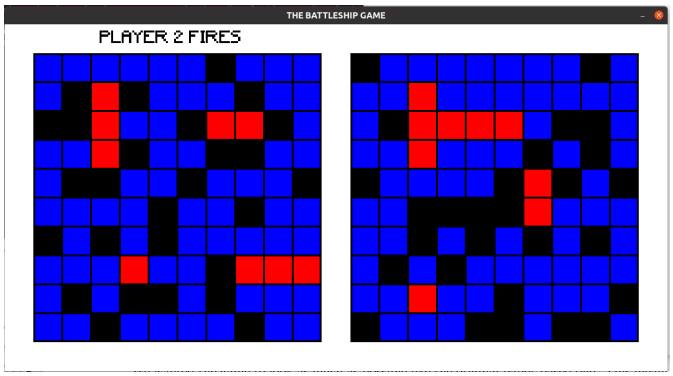


Once both players are done with the positioning, the first phase of the game is over, and both grids appear back on the screen, with all the cells showing the same color (blue). Now the attack phase can begin. We wanted to find a way to replace the row-column declaration of the original game with a faster mechanism. Thus, we decided to let the user click on a cell of the opponent's grid to simulate an attack on that cell. Furthermore, we needed a way to show whether the attack was successful (hit) or not (miss). We used colors to do so: red cell stands for a "hit", and black cell for a "miss".





At this stage, players should have to take turns in shooting at each other, and the program should remember (and display) all the hits and misses. Furthermore, the program prevents a user from mistakenly clicking on an already shot cell, on one of his own cells, or even outside of the two grids.



The program does also keep track of how many ships a player has left, and it ends the game whenever a player has lost all of them. At this point, the score of the players is updated, and an end-of-game message is printed on the screen. The players are then given the chance to play a new game, without losing count of the overall score of the previous games.

Design Decisions

In the following paragraphs, the design steps which are common to all three versions will be presented. Then, three subsections will outline the design choices specific to each version.

We decided to build this game on top of the Pygame module. Pygame has a **display** object that can be used as the "canvas" for the game. The first step in the making of this program is to initialize this. Therefore, a display object is created, with certain width and height measurements: 1380x720. This ratio was specifically chosen to enable us to use 60x60 pixels cells for the grids.

We wanted this program to be able to run on any machine, independently of the CPU. Therefore, the **time.Clock** module was used so that the display can be updated at a specific frame rate (which otherwise would be the maximum rate that the CPU can handle). We found that a suitable number was 30 FPS.

Using the so-defined clock, the display is redrawn at every clock tick. This allows us to visualize all the game events. For instance, when a ship is moved on the grid, a new clock tick will redraw the display showing the ship in the new position (while deleting the ship which was previously drawn). When a ship is hit, at the next iteration the corresponding cell would be redrawn using the color red, et cetera.

We used a dictionary to conveniently store the position of each cell on the grid as well as its state (untouched, hit, miss). The key to this dictionary is the tuple of the coordinates of this cell, while the value is its color (tuple of RGB values). Blue for untouched, red for hit, and black for missed.

Version 0

This version had the goal of letting us develop and test the main functions of the game before deploying them on the actual 2-players version.

We wanted to develop a version of the game where the user plays against the machine. Indeed in this game, the program randomly places five ships on the board. Then, the player has a total of 50 shots available to sink all the ships. The game is won if all the ships are sunk (and a win message is displayed).

The first action that this program performs upon its execution is to create a list of all the cells containing a part of a ship. By calling the **getShips()** function, a list is obtained by randomly choosing a fleet of ships from a json file where a set of different fleets is stored. The **ships** list will contain a list of tuples, each one representing the grid coordinates of a cell that contains a piece of a ship, and should thus be represented as red if hit.

The next step is to create a dictionary for the grid, where each key represents a cell (actually its coordinates) and the value is the color of the cell. At the beginning of the game, since no shot has been fired yet, all the cells will be blue by default. This is achieved by calling the function **initColorDict()**.

Once the ships are positioned and the grid is printed on the display, the player can start guessing. A while loop is used to redraw the screen based on the user's input. Specifically, at each iteration, the program looks for clicks on the grid using the **clickCell()** function, and checks whether the shot is valid (i.e. shot is on a blue cell). If it is, at the next iteration the cell will turn red or black, and the counter of available ammunition will decrease by 1.

Furthermore, the program keeps count of the remaining ships and bullets, whenever one of the two becomes 0, a win or a loss message is displayed using the **displayMessage()** function and the game is terminated.

Version 1

This version of the game is an intermediate step toward the development of the final version. Here, two players can play a game against each other. This program is built on top of the previous one, recycling most of its functions.

The first action that the program carries out is to initialize the cells' coordinates and colors dictionary. It does so by calling the **initColorDict()** function twice. Of course, at the beginning, all cells are set as blue (untouched).

At this stage, the two grids, as well as the cells, can be drawn on the screen. This is done using the **drawCells()**, **drawLeftGrid()**, and **drawRightGrid()** functions.

Now each player takes turns positioning his ships. The function **placeShips()** is called for Player-1. This function contains four **while loops**, one for each of the ships to be positioned. Each ship is spawned at the upper-left corner of the grid (or at the closest ship-free cell), i.e. certain cells of the grid become red. Then the while loop looks for the user input (keyboard). If an arrow is pressed, the position of the ship on the grid has to be updated accordingly. So, first, the cells are turned blue to delete the previous ship, then the new ship is drawn by turning red the new cells. At the same time, checks are in place to prevent the player from moving the ship outside of the grid. Once the player is happy with the position of a ship, he can press enter. The program will detect this and break the loop. The position of the ship is saved in the **ships set**. This set is needed to keep track of where the already placed ships are, so that the program does not allow the user to place a ship on top of another ship. In the end, this function returns the updated dictionary containing all the positions and colors of the cells. The variable **player_1_cells_colors** is assigned to this dictionary.

After this stage, all Player-1's ships will be drawn on the screen. They must of course be hidden from the sight of Player-2 before he can position his ships. This was one of the hardest design decisions of the project.

We decided to take care of this in the following way: we initialized a new cells-colors dictionary, called **guess_cells_p1**. This way, we could use the former dictionary to store the coordinates of the ships, while the new dictionary is going to be used just for visualization purposes, and can thus be modified during run-time depending on the shots of the players.

Once this is done for Player-1, the new dictionary can be used to "sink" all his ships. The functions **drawCells(guess_cells_p1)** and **drawLeftGrid()** are used to cover up the grid of Player-1 with blue cells.

At this stage, Player-2 has to position his ships, and the same functions are called. Once he is done, all the ships are sunk again, and the attack phase can begin.

This phase is handled using a while loop. The first thing that is done inside this loop is to call the **clickCell()** function with the variable **guess_cells_p2** as argument. This function is taken from version-0. It waits for a click of the user on a cell of the opponent's grid, then returns the coordinates of that cell. This cell is then checked against the **player_1_cells_colors** dictionary, to see whether the shot results in a hit or a miss. The color of the cell is modified accordingly by updating the **guess_cells_p2** dictionary. Then, also the **player_1_cells_colors** dictionary has to be updated, and the cell is turned to "sunk". The reason why this is done will be explained later. Once this first part is over, the screen has to be redrawn, and the usual functions are used to do that.

It is now time for Player-2 to carry out his attack. This phase is handled using a very similar code to the one for Player-1.

After each attack is over, it is important to check whether there is a victory. A victory happens when a player has sunk all the ships of the opponent. Thus, a function is defined to check how many ships a player has left. This function is called **countShips()**. It counts how many red cells are there in each cells-colors dictionary. This is why it is important to change each red cell to "sunk" once it is hit: so that it is not counted as red. When this function detects that a player has no red cells left, it posts a **pygame.event**. Two custom events were defined: one for the victory of Player-1, one for the victory of

Player-2.

At this point, the code inside the while loop reaches the end and so it is reiterated, and the first action which is carried out is to check whether any event was posted. If the victory of any player is detected, the game has to stop. This is done using the **displayMessage()** function, which will redraw the screen with a message telling the users which player has won the game.

Version 2

Although version-1 already allowed two players to play a game, it still did not meet all the specifications. In particular, it only allowed the players to play one game, while we wanted the players to be able to keep on playing as many games as they wanted (while keeping count of the overall score).

In order to achieve this, a new version was developed, building on top of the previous one. In this version, we implemented three main updates, based also on feedback from some friends of ours who tried out the game:

- First, we addressed the problem that new users might face when playing their first game. Specifically, we added on-screen instructions to guide the players in positioning the ships.
- Next, we added some text on the screen to indicate clearly which user's turn it is, i.e. who should be shooting now.
- Finally, we added a score counter, which is updated after every game, as well as a "play again" button.

In order to address the first issue, we developed a function called **drawRules()**, which guides the user in positioning the ships on his grid. This function gets called at the beginning of the **main loop**. It uses Pygames modules to draw some text on the screen at a specific position and with a certain font. Following that, the code is the same as version-1 for the positioning phase.

When the attack phase begins, a new function called **drawTurn()** is used to print a message on the screen telling the players whose turn it is to take a shot. The rest of the code for the attack phase is the same as for the previous version.

We needed a way to keep track of and update the overall score of all the games played. This could simply be done using a list containing two elements, which the **score** variable was assigned to and initialized to [0,0] outside of the main function. Whenever a win-event is posted, this variable is updated accordingly.

Finally, when a victory is detected, the players should be given the chance to play a new game. A victory message is then printed on the screen, as well as the current score. Furthermore, a play-again button can be pressed. If pressed, it issues a call to the main function, so that all the game variables (except for the score) are reinitialized. A new game can then begin. This button was implemented using some functions from the Pygame module, which are defined inside the **displayWinner()** function.

Debrief

During the development of this program, we had to tackle several issues. As previously mentioned, one of the hardest problems to solve was how to handle the visualization of the ships on the screen while remembering the original position where the player placed them. We believe that the solution we came up with (using two coordinates-colors dictionaries) is very effective, as we could use the same data structure for those two different purposes.

Furthermore, we found that it was difficult to let the user move around the ships on the board during the first phase. It took some time to think about all the if-conditions to check so that no ship went outside

of the board, and the code we ended up writing is pretty long and redundant. Anyhow, it runs well and it does what it is supposed to.

In the end, we met all the specifications that we set for our project, and we are particularly satisfied with the results. Nevertheless, there are still several features that we could add in future developments. For instance, we could develop a new ships-positioning method, making it drag-and-drop. This could probably be more comfortable for the user to use, thus it would improve the user experience. We would also like to add a leader-board which is displayed at the end of every game. This would keep track of the players' usernames, and all their victories. It could be stored as a dictionary inside a json file. Another interesting feature we could work on is the graphics. We could use drawings of ships instead of squares, and make the rest of the game feel more realistic too, perhaps using sound effects. We believe that all this can be done using Pygame. The biggest improvement we would like to make, however, is to allow users to play this game on two separate machines using a local Wi-Fi (we believe this can be achieved using, for example, the **networkzero** library.)

Conclusion

In conclusion, we found this project to be rather challenging, but it definitely allowed us to learn more about the core mechanisms of Python (and the Pygame module specifically). Our knowledge and understanding of this language was put to the test, and this forced us to challenge what we believed we knew and to learn it again if necessary. Often we found ourselves studying again basic Python functionalities. Of course, we also had the chance to learn about more advanced topics and libraries. All things considered, this project gave us the chance to deepen our knowledge in this field. Moreover, it was really satisfying to build something that works and that we can have fun using.