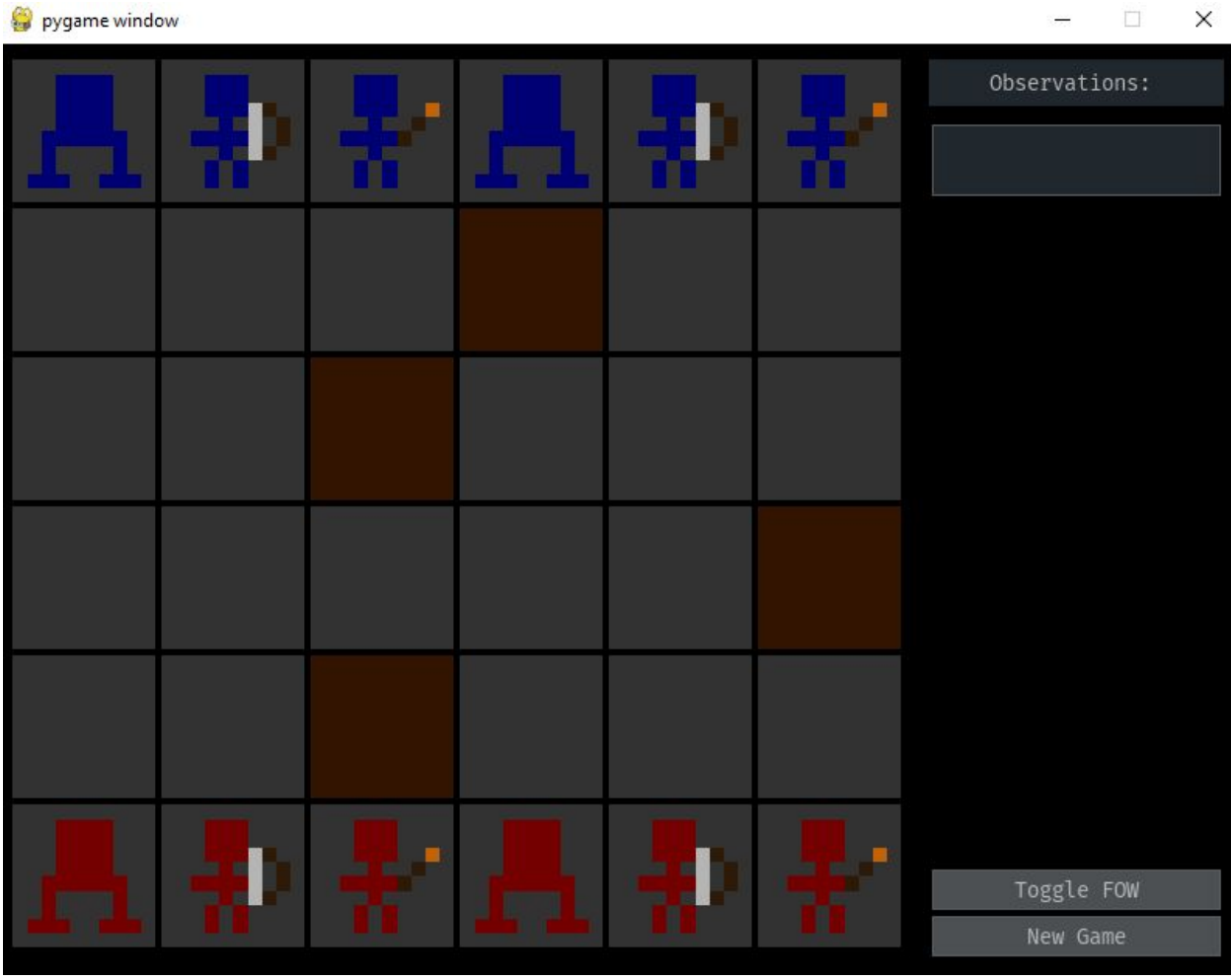
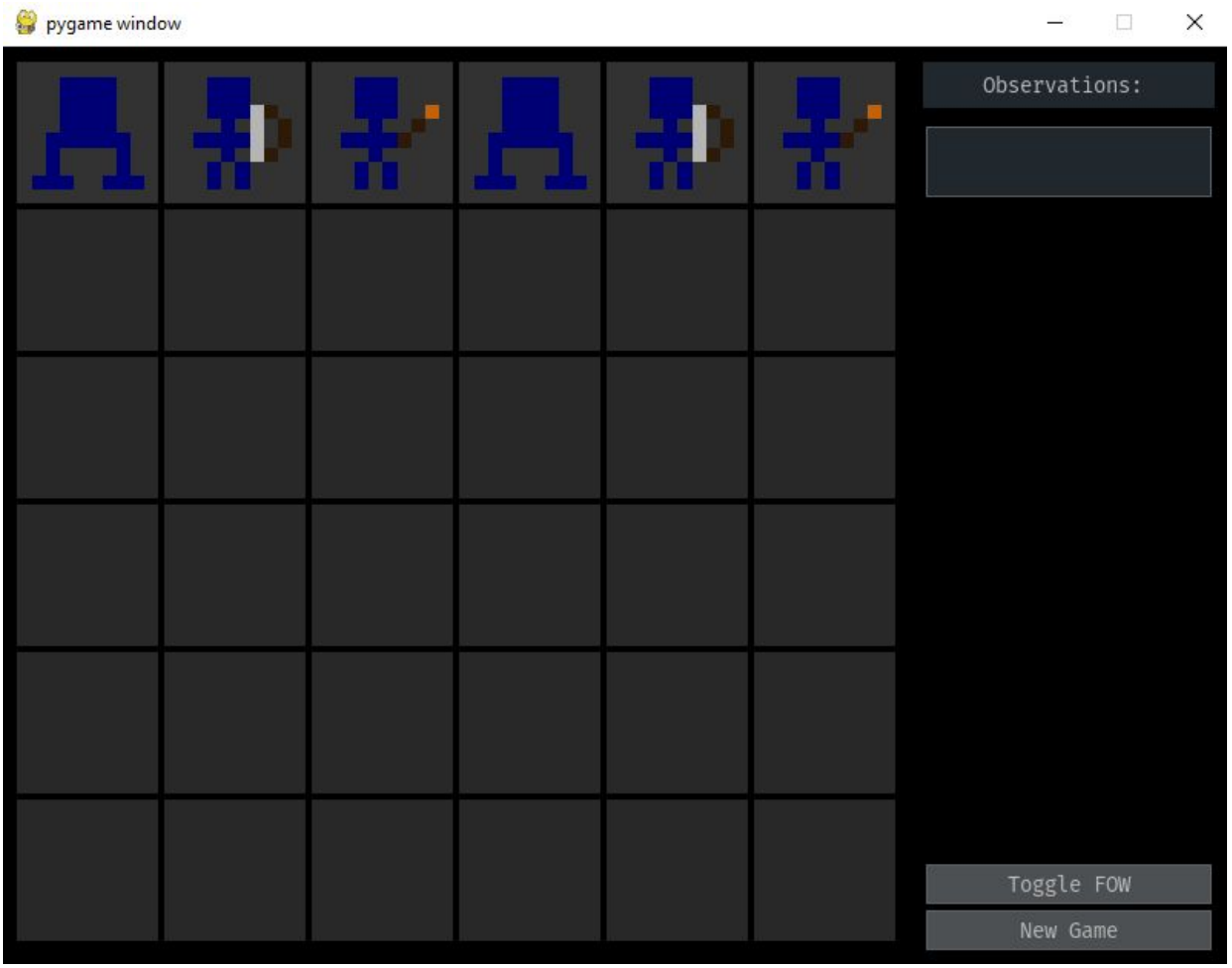
1. Boolean
    a. Established booleans for each cell on the board
2. Fog of War
    a. 

        i. The observations box in the top right displays a breeze or stench in the appropriate situation. As can be seen in the bottom right, there is a “Toggle FOW” button, which when clicked hides the opponent’s pieces and the pits as shown below.

b. 

3. Probabilities
    a. Started off each probability based on where the opponent's pieces are situated (i.e the opponents first cell is a wumpus, and so on). The cells in the middle of the board were all equally likely of being pits. From then on, the probability was recalculated after each move of the opponent (The AI) and after each move by the human player.

i.

```
def update_after_opp(c, prob, neighbors):
    d = len(prob)
    new_prob = prob

    # Assumes that the opponent has an equal chance of moving any pieces
    for i in range(d):
        for j in range(len(prob)):
            new_prob[i][j] = (1 - 1 / c) * prob[i][j]
            sum = 0
            for x in range(max(0, i - 1), min(d, i + 2)):
                for y in range(max(0, j - 1), min(d, j + 2)):
                    # Adds to sum
                    sum += prob[x][y] * 1 / (c * neighbors[x][y])
            new_prob[i][j] += sum

    # Normalize the probability to ensure Bayes rule is satisfied
    prob = normalize_prob(new_prob, c)
    return prob
```

ii.

```
def update_after_player(occupied, prob_table, size):
    # units is a string of all of the observations
    # max_units is a dictionary of the max number of pieces that can be adjacent to one of our units
    # adjacent is a list that will contain all the adjacent cells to an occupied cell that yields an observa
    units = ""
    max_units = {"b": 0, "s": 0, "n": 0, "h": 0}
    adjacent = []
    # First finds all of the occupied cells that yield observations
    for (x, y) in occupied:
        # Observe should return one of the following: "b", "s", "n", "h", or ""
        observation = board.Board.observe(x, y)

        # If there is a relevant observation, add all of its adjacent cells to the adjacent list
        if observation != "":
            for i in range(max(0, x - 1), min(size, x + 2)):
                for j in range(max(0, y - 1), min(size, y + 2)):
                    adjacent.append(i, j)
            # Append the observation itself to units
            if observation == "b":
                units += "p"
            elif observation == "s":
                units += "w"
```

4. Probability-Based Move
    a. In order to be able to make a move, some assumptions need to be made. We are operating under the assumption that the highest probability for any given cell is what is in that cell. So, if a cell has the highest probability of being a wumpus over anything else, it is assumed to be a wumpus. If it has the highest probability of being a pit, it is assumed to be a pit. This allows us to progress with some type of certainty, and the agent can make its move as it normally would, with this assumption-based idea of what the board looks like.
5. Better Alternative
    a. As a better alternative to assuming a random move by the agent, we have integrated the probabilities into minimax. We try to determine which piece the opponent will move based on our heuristics for minimax. Similar to what the agent does in number 4, the algorithm also assumes that the highest probability for any given cell is what is in that cell. From there, the minimax algorithm is run,

and it is predicted what the AI move will be. After that prediction is made, the probabilities are updated assuming that is the move that was made.

6. Final Implementation
    a. The final implementation is done via minimax as discussed.