Minesweeper

By Ryan Lo
704579791

PIC 16 Final Project
Fall 2018

Introduction

Minesweeper is a popular single-player puzzle video game that has spawned many variations, it’s most popular version pre-installed on Windows XP.

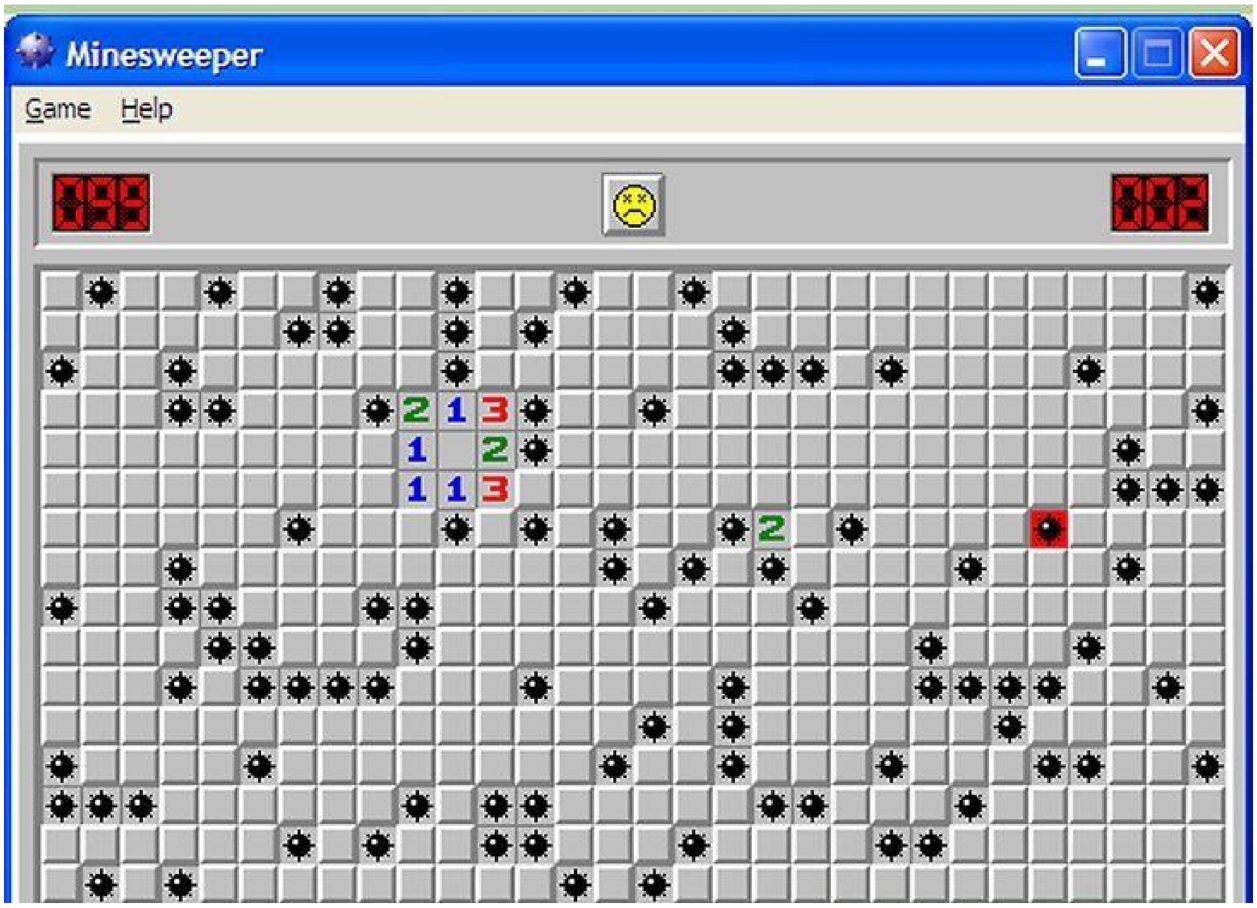

Figure 1: Original Minesweeper game on Windows XP

The objective of the game is to clear the rectangular board of all its hidden mines without accidentally detonating any of them. The player does this by clicking on the grid’s squares to reveal numerical clues as to how many mines surround the given square. A flag can also be placed on the square to help the player keep track of which squares he or she thinks might contain a mine (regardless of whether or not it actually has one, and will not trigger the mines). The game ends when the player has either clicked on a hidden mine (in which the player has lost) or has successfully cleared the grid of all mines by placing a flag on them.

Problem Definition

In this project, we have to design and implement the Minesweeper game in Python. The game should be rendered in a GUI module in Python to display each grid cell in the board (a 2D grid layout). To make the game less predictable, the game grid should randomize the locations of the mines placed. The player should be able to left-click to open a square in the grid and right-click to place or remove a flag in the grid cell he or she clicks on, and that square should be updated with either the number of mines hidden in the neighboring squares, or reveal the locations of the mines if it contains a mine. If the player successfully completes the game (by placing all the flags on all the mine locations, which there are 10 in this implementation for our 10x10 default grid) or fails and trips a mine, a "Game Over" dialog box should alert the user accordingly. Additionally, should the player click on a square with zero neighboring mines, all other adjacent squares containing zero neighboring mines should automatically be revealed. Lastly, there should be two labels outside of the board that update depending on the game's state: one to indicate the amount of time has passed in seconds (to let the player know how long they've taken to complete the game so far, which they're trying to minimize), and the number of flags they have left to place. In my version of the game, orange squares represent flags placed, while red squares are the locations of mines (not marked by flags) and mine locations upon end of game will be revealed with a black circle in the grid.

In the following pages, I will outline implementation of the aforementioned problem definition with the objective of guiding any experienced Python programmer how to create the game from scratch.

Minesweeper Implementation

0. TkInter Library

The following section will detail the various components of the project and describe how they are implemented in Python at a high-level. The entirety of this Minesweeper game's GUI was implemented using Python's TkInter library. Like most GUI library modules, graphics can be represented as an MxN pixel canvas to draw colors, shapes, and more. TkInter represents most of these with layers of "widgets."

I. Main Function

With the main function, we need to declare a new TkInter root that will create the top-most level TkInter widget for our application, the root. It then uses the root to initialize a MainScreen class object, which houses all the logic for our Minesweeper game, which will then run and update the game board view. Additionally, MainScreen is also initialized by passing in a value *n=10* as a parameter because this is the length of our square game board: a 10x10 grid. This parameter would allow us to pass in other user-input values to customize the board's dimensions, given more time to finish this project. Below is a high-level implementation of *main()*.

```
function main():
        root = top-most level TkInter widget for our app.
        Create new MainScreen object, giving it `root` and
             `n=10`
        Have root mainloop to keep the app running until we
             close it.
```

II. MainScreen Class - Part 1: Game Board Initialization

As mentioned earlier, the MainScreen class houses the general logic of drawing the Minesweeper board on screen and managing the player's progress in the game. For this section, we will only examine the part of MainScreen that initializes and sets up the game board upon user entry. Unlike most basic Python classes that extend *object*, MainScreen extends *tk.Frame* to be able to be used as the main TkInter GUI display for our game.

MainScreen class has several useful properties and fields:

1. An integer, *n*, to represent the number of rows and columns our NxN grid has
2. An integer, *length*, to represent the length of each square in our game grid
3. An nxn 2D-array, *grid*, to represent our game board. It will contain GridCell objects (explained later).
4. A dictionary, *mines*, to map *(r, c)* integer tuples (zero-indexed row-column coordinates) to booleans for quick lookup of the row-column locations of the mines in *grid*.
5. An integer, *flags*, to indicate the number of remaining flags the player can place in the grid. The player starts with *n* flags, for *n* mines.
6. A boolean, *gameover*, that records the state of whether or not the game is over.
7. An integer, *sec*, to represent the number of seconds the player has taken to attempt the game
8. A TkInter canvas, *canvas*, object that will be used to draw the objects on screen.

The MainScreen class also has several useful functions it can call to set up the game. Below is the documentation of several such functions. Note that some functions are referenced that will be explained more in Part II of MainScreen functionality. :

1. *__init__(parent, n)* - this is the constructor of MainScreen. It takes in the TkInter root object (see main function) for set up with the TkInter library functionality, and an integer to denote the length of our square NxN grid. In this case, N is 10.

```
        __init__(parent, n):
             Set up tkInter canvas (I used 1000px by 800px)
             Bind left mouse click to MainScreen function that
handles action of clicking a cell (here, it's clickCellAction).
```

```
               Bind right mouse click to MainScreen function
that handles action of placing a flag (here, it's
clickFlagAction).
               Call initGame function to set up board
               Call drawGame function and drawTime function
```

2. *initGame*() - this function initializes all the non-tkInter properties of MainScreen with default values, which include: *n* (input n value)*, length* (input length value)*, grid* (empty NxN array)*, mines* (empty dictionary)*, flags* (*n*)*, gameover* (false)*, sec* (0). It then goes on to call *initGrid()* set up the grid with mines, and then draws the time outside the board with *drawTime()*.

```
          function initGame():
               Initialize all variables as mentioned earlier
               Call initGrid()
               Call drawTime()
```

3. *initGrid()* - this function sets up our *grid* array to have NxN GridCell objects, then randomly places the N mines in N of those GridCells (via Python random module), and finally for each GridCell counts the number of neighboring squares that contain a mine (by essentially checking all 8 adjacent squares).

```
          function initGrid():
               Loop N times:
                    Generate (r, c) random coordinate
                    If (r, c) not in mines, add (r,c) to mines dictionary.
Otherwise, redo this iteration again
               Loop through each index (i, j) of NxN grid:
                    Add to grid[i][j] a new GridCell object
                    If (i, j) in mines dictionary, update GridCell's
hasMine property to True.
               Loop through each index (i, j) of NxN grid:
                    Look through all 8 adjacent neighbors of (i,j) index:
                         If neighbor hasMine, increment GridCell at
(i,j)'s surroundingValue.
```

III. GridCell Class

The GridCell class is a class to represent all the squares in the Minesweeper game grid. It is primarily useful to hold a particular square's information useful for the state of the game. In particular, it holds several class properties:

1. Two integers, *x* and *y,* that represent its horizontal and vertical pixel coordinates (where it will be drawn on the canvas)
2. An integer, *size*, that represents length (which will be used to form a square in the board)

3. A boolean, *clicked*, to indicate whether or not the user has clicked on this GridCell yet
4. A boolean, *hasFlag*, to indicate whether or not the user has placed a flag on this GridCell
5. A boolean, *hasMine*, to indicate whether or not this square has a hidden mine in it
6. An integer, *surroundingValue*, that represents how many mines are in the eight squares adjacent to this particular GridCell.

The GridCell object also contains two useful functions that the MainScreen class will call to check the GridCell's properties when it goes to modify the grid.

1. *__init__(x, y, length)* - this is the constructor of GridCell. It takes in the x and y Cartesian coordinates of the GridCell and the length of the cell's square body. It then initializes all its respective properties, as well as the other ones with default values.

```
__init__(x, y, length):
     Set x with input x
     Set y with input y
     Set length with input length
     Set clicked to be False
     Set hasFlag to be False
     Set hasMine to be False
     Set surroundingValue to be 0
```

2. *getCoordinates()* - this function takes no input and returns a 4 length tuple of the Cartesian coordinates of the GridCell's top-left and bottom-right corners in the form of *(x_1, y_1, x_2, y_2)* where *(x_1, y_1)* represents the top-left and *(x_2, y_2)* represents the bottom-right. This is calculated by making a tuple with the first 2 coordinates the *x* and *y* position properties aforementioned, and then making the last 2 coordinates these same 2 coordinates but shifted by the *length* property of the square.

```
function getCoordinates():
     x1, y1 = this GridCell's x and y positions
     x2, y2 = this GridCell's x and y positions
shifted
          by its length.
     return tuple ordered x1, y1, x2, y2.
```

3. *checkClick(ev)* - this function takes in an event object, ev, (passed in from the TkInter root when a mouse button is pressed. It checks if the mouse's click coordinates are contained in the GridCell's boundaries (via its location coordinates and its length) and returns a boolean to indicate so.

```
function checkClick(ev):
     coords = this GridCell's coordinates
          (getCoordinates)
          If ev’s x and y coordinates are outside the box
     formed by coords, return False. Otherwise return True.
```

Below is the general API of the *GridCell* class.

```
     class `GridCell` inherits from `object`:
          Properties:
               (int) x
               (int) y
               (int) length
               (boolean) clicked
               (boolean) hasMine
               (boolean) hasFlag
               (int) surroundingValue
          Functions:
               __init__(x: int, y: int, length: int)
               getCoordinates(): tuple(int, int, int, int)
               checkClicked(event: TkInter event): boolean
```

IV. MainScreen Class - Part 2: Game Logic

Once the game board has been initialized (all GridCell objects are in place, all fields set to default starting values, and the mines are randomly placed), it is time to implement the logic for when the player interacts with the game. Note that this section will focus on several of the key functions; there are many implemented functions for this part that are rather small and will only be mentioned in the API of MainScreen because they are rather self-explanatory and can easily be implemented in very few lines of code.

1. *drawGrid()* - this function effectively draws the game grid from the top left corner to about 3/4ths of the width of screen and the full height of the canvas. It does so by looping through *grid* and drawing a TkInter rectangle (specifically a square) on the canvas at the cell’s Cartesian coordinates of length 100. Before drawing, however, we want to check the cell’s information to decide what color and text to display in the GridCell if any. If the cell was clicked and doesn’t have a mine, the color of the cell is gray and the cell’s surroundingValue is placed in the middle of the box. If the cell wasn’t clicked, simply set the color to light gray. If the cell has a flag on it, set the color to orange. If the cell has a mine and the game is over, set the background color to red (unless there’s a flag in it) and draw a black circle on top to indicate a mine. Note that because TkInter only draws shapes on top of existing pixels, we need to redraw the grid on top of any existing drawn grid to cover it up so that the user has a fresh display of the grid. This is to give the

illusion to the player that the state of the game has updated while working around the restrictions of TkInter.

```
function drawGrid():
    For each GridCell in grid:
        If cell was clicked and hasMine not true:
            Draw square on canvas of color gray at that cell's coordinates.
            Draw text in middle of square containing cell's surroundingValue
        Else if cell has mine and game is over:
            If cell doesn't hasFlag, draw square on canvas of color red at that cell's coordinates. Else, draw square on canvas of color orange.
            Then draw a black circle on top.
```

2. *drawTime*() - this function effectively draws a white rectangle with text displaying the time in seconds how long the user has spent so far playing the game in the upper right region of the canvas. Because TkInter only draws shapes on top of existing pixels, we need the rectangle to cover any time text from previous drawings so that the text doesn't get blurred, and then we draw the text on top of a fresh new clean background. After drawing the text, if the game is still not over, we can then increment *sec* by 1. To call this function again after another second passes, we can use the tkinter root to create a callback function so that after 1000 milliseconds (1 second), it calls drawTime again.

```
function drawTime():
    Position coordinates of our text in upper right off grid.
    Draw on canvas a white rectangle at those coordinates, followed by the text displaying the string "Time: x".
    If the game is not over:
        Increment sec
        Use root.after() to trigger drawTime again after 1000 ms.
```

3. *clickCellAction(ev)* - this funciton is run every time we left click. It takes as input a TkInter click event, *ev*. We check if ev's click coordinates happened to land in the boundaries of a GridCell object. If so, we effectively click into that cell. We set its *clicked* value to true and then check the cell's contents; if it contains a mine, set gameover to true and trigger the lose game message box. If it had a flag, return and dont do any more. If it had an empty cell (zero surroundingValue of mines) then clear out the empty cells nearby. Then redraw the grid (this effectively just calls the drawGrid(), etc

functions to draw the new GridCells, time, and flag value boxes on top of the existing canvas to cover the old shapes on the board.

```
      function clickFlagAction(ev):
            If game is over, do nothing
            Loop through all cells in grid:
                  If cell.checkClick passes:
                        Set cell.click to true.
                        If cell hasFlag:
                              Return (do nothing)
                        Else if cell hasMine:
                              Display message box player lost.
                        If cell has surroundingValue of zero:
                              Call clearEmptyCells on current
(row, col).
                        Break out of loop and redraw grid:
                              drawGrid() and drawFlags()
```

4. *clickFlagAction(ev)* - this function is run every time we right click. It takes as input a TkInter click event, *ev*. We check if *ev*'s click coordinates happened to land in the boundaries of a GridCell object. If so, we attempt to place a flag there, decrement the number of flags left, and redraw the canvas board if need be with the appropriate flag value and marker in the space, as well as check if we won the game. If we right clicked on a space already with a flag in it, we want to take away that flag and increment the number of flags left, and redraw the game accordingly.

```
      function clickFlagAction(ev):
            If game is over, do nothing
            Loop through all cells in grid:
                  If cell.checkClick passes:
                        If cell hasFlag:
                              Set cell hasFlag to false
                              Increment flags
                        Else:
                              Set cell hasFlag to true
                              Decrement flags
                              Call checkWinGame()
                        Break out of loop and redraw grid:
                              drawGrid() and drawFlags()
```

5. *clearEmptyCells(n, row, col)* - this function is called whenever the player selects a square at (row, col) containing a surroundingValue (number of mines in adjacent squares) of

zero. It effectively runs depth first search to iteratively clear all neighboring squares in all directions that also contain zero surrounding mines, and sets them as clicked by user.

```
          function clearEmptyCells(n, row, cell):
               If (row, cell) not in grid boundaries: return
               Cell = get GridCell from grid at (r, c)
               If Cell has a mine, was clicked, or its
surroundingValue is not zero: return
               Otherwise, remove flag from cell if placed, set
cell.clicked as true.
               Then call clearEmptyCells in cell north, then
south, then east, and then west to current cell.
```

6. *checkWinGame()* - this function checks to see if the user has won the game by scanning all N items in *mines* dictionary and seeing if the GridCell objects at those spaces (they contain mines) and see if they all also have a flag placed on them. If one or more don't have a flag on them, return False. Otherwise, we have won the game and return True.

```
          function checkWinGame():
               Loop through all (r,c) keys in mines:
                    Cell = get GridCell object at (r,c) in grid
                    If Cell hasFlag isn't true, return False
               Finished looping and all flags set, so return
True after displaying TkInter message box informing user they
won and how long it took to finish.
```

Below is the full, finished API documentation of the MainScreen class:

```
     class `MainScreen` inherits from `tk.Frame`:
          Properties:
               (int) n
               (int) length
               (2D list of GridCells) grid
               (tuple(int,int) dictionary) mines
               (int) flags
               (boolean) gameover
               (tk.canvas) canvas
          Functions:
               __init__(root: tk.root, n: int)
               initGame(): void
               initGrid(): void
               drawGrid(): void
               drawTime(): void
```

```
clickCellAction(ev: TkInter event): void
clickFlagAction(ev: TkInter event): void
clearEmptyCells(n: int, row: int, col: int): void
checkWinGame(): void
```

Conclusion

This concludes the report containing the descriptions, documentations, and brief implementations of the main functionalities of the Python TkInter adaptation of Minesweeper. There are a few more small steps not explicitly described in this report that I used to fully implement this project, but they are fairly easy to figure out. Furthermore, I believe these steps are everything one needs to begin work on their own Python version of Minesweeper and fully implement its features and more.

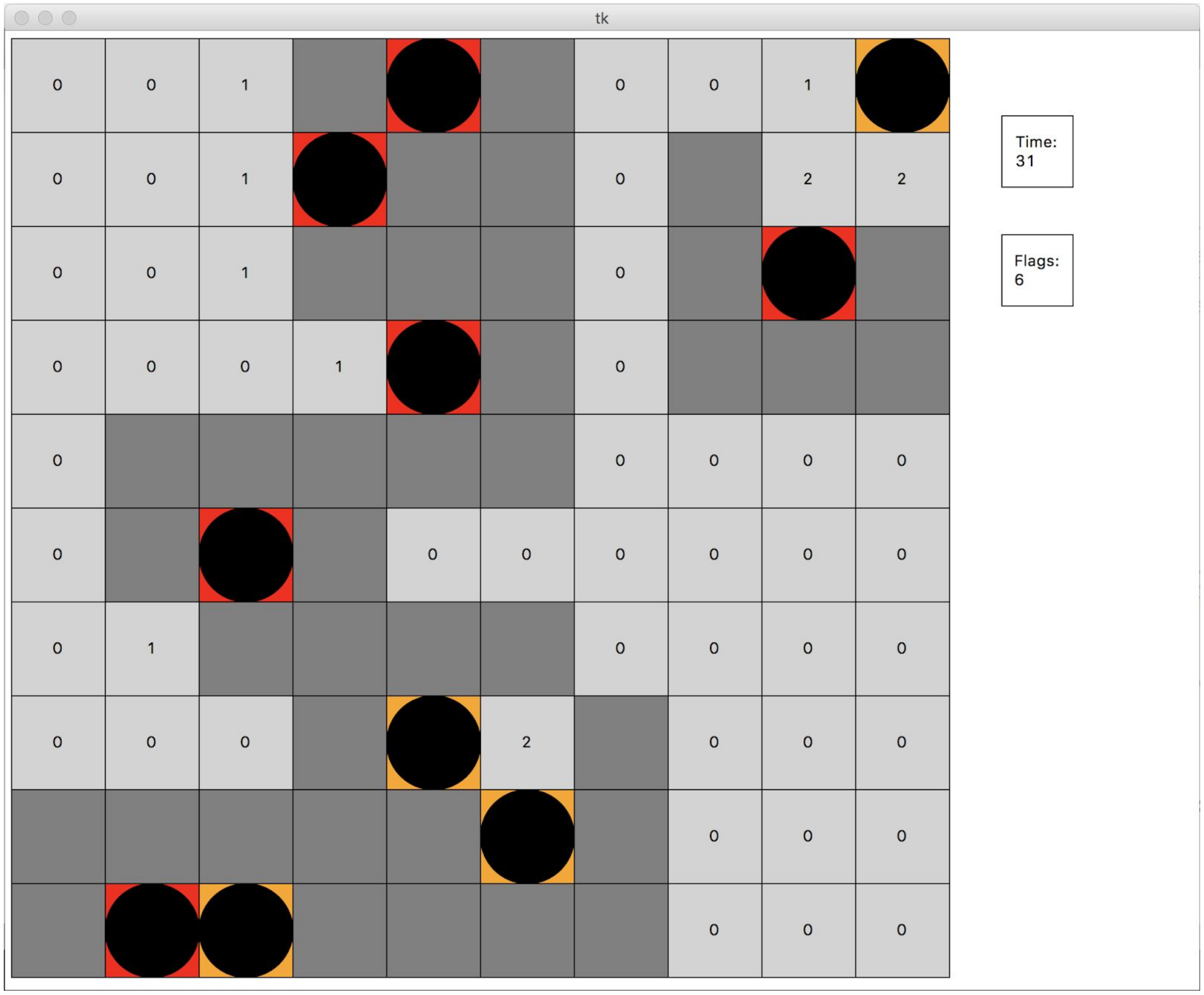

Figure 2: Final product of the Minesweeper game.
Currently, the game is over and the user has lost