CSSE1001 Assignment 3

Due 5pm, Friday 29th May

1 Introduction

In Assignment 1 you implemented a text-based game, *Pokemon: Gotta Find Them All!*. In Assignment 3 you will extend this game to a graphical user interface (GUI) based game. Your implementation should adopt a model-view-controller (MVC) structure, similar to Assignment 2.

The new version of this game, *Pokemon: Got 2 Find Them All!*, is a single-player GUI-based game in which the player is presented with a grid of 'tall grass' squares. Some tall grass squares hide Pokemon, and some do not. The aim of the game is to 'catch' all Pokemon by right-clicking on the tall grass under which they hide, **and** to clear a safe path for other trainers by left-clicking on the tall grass squares in which Pokemon are not hiding. Left-clicking on a tall grass square in which a Pokemon is hiding will scare the Pokemon into battle, and as your player has ventured out without any Pokemon of their own this will cause them to lose the game. To assist your player on their quest, when a blank tall grass square is revealed the number of Pokemon in adjacent squares should be displayed on that square.

2 Tips and hints

The number of marks associated with each task is not an indication of difficulty. Task 1 may take less effort than task 2, yet is worth significantly more marks. A fully functional attempt at task 1 will likely earn more marks than attempts at both task 1 and task 2 that have many errors throughout. Likewise, a fully functional attempt at a single part of task 1 will likely earn more marks than an attempt at all of task 1 that has many errors throughout. While you should be testing **regularly** throughout the coding process, at the minimum you should not move on to task 2 until you have convinced yourself (through testing) that task 1 works relatively well, and if you are a postgraduate student, you should not attempt the postgraduate task until you have convinced yourself (through testing) that task 1 and task 2 both work relatively well.

Except where specified, minor differences in the look (e.g. colours, fonts, etc.) of the GUI are acceptable. Except where specified, you are only required to do enough error handling such that regular game play does not cause your program to crash or error. If an attempt at a feature causes your program to crash or behave in a way that testing other functionality becomes difficult without your marker modifying your code, comment it out before submitting your assignment.

You may import any standard libraries, but you must not make use of third-party libraries.

You must write all your code in one file, titled `a3.py`. Your game must display (in the latest attempted mode) when your marker runs this file.

3 Task 1: Basic Gameplay - 10 marks

Task 1 requires you to implement a functional game of *Pokemon: Got 2 Find Them All!*. Squares will be represented by rectangles on a canvas. Different cell states are portrayed via the colour of the rectangle (dark green for 'tall grass', light green with the number of surrounding pokemon for 'short grass', red for 'attempted catch', and yellow for 'exposed pokemon'). Figure 1 gives an example of the game at the end of task 1.

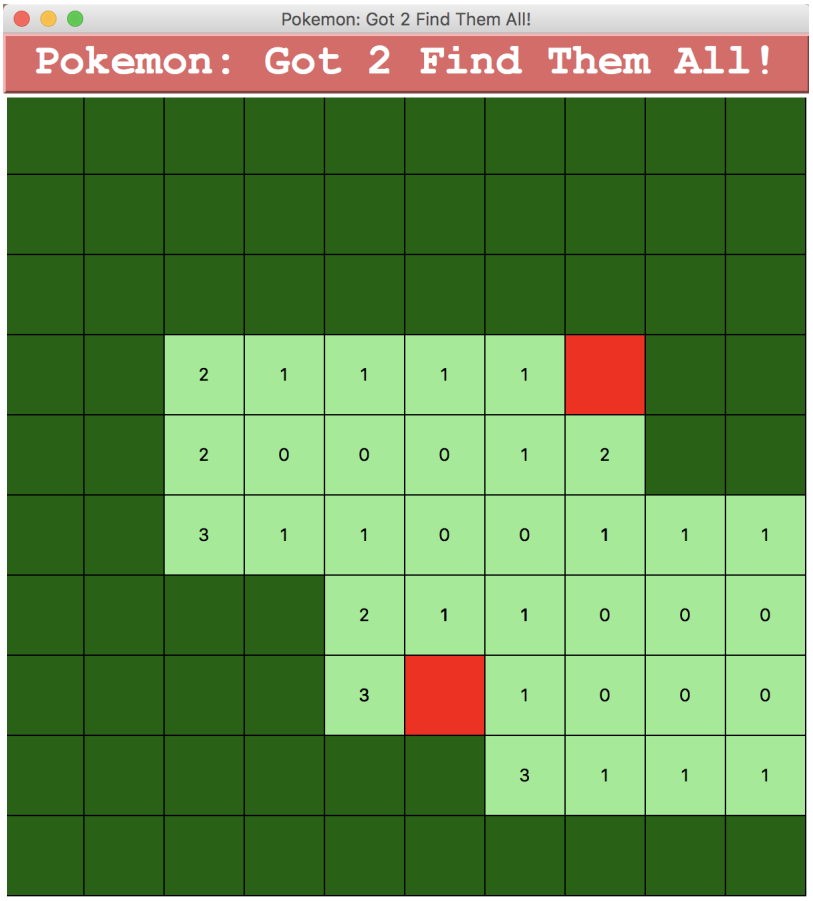

Figure 1: Example game at the end of task 1.

In order to complete this task, you must implement the model for gameplay by adapting Assignment 1 code into a class, implement a view class for the game board, and link the two with an overall controller class. The following sub-sections outline the required structure for your code. You will benefit from writing these classes in parallel, but you should still test individual methods as you write them. Upon game completion (i.e. win or loss), the user must be informed of their result via a tkinter messagebox.

3.1 BoardModel

The `BoardModel` class should be used to store and manage the internal game state. This class must be instantiated as `BoardModel(grid_size, num_pokemon)`, where `grid_size` is the number of rows (equal to the number of columns) in the board, and `num_pokemon` is the number of hidden pokemon. Many of the functions created in your Assignment 1 may be relevant to adapt to methods here. You are permitted to use any support code or sample solutions (course provided) from Assignment 1, as well as any of your own code from Assignment 1. You must not, however, use any code from the Assignment 1 submissions of other students. Additional methods that may

be useful to write for this class are described in Appendix A. You may also find it useful to add your own methods.

3.2 PokemonGame

`PokemonGame` represents the controller class. This class should manage necessary communication between any model and view classes, as well as event handling. You must also write code to instantiate this class and ensure that the window appears. Give your window an appropriate title, and (as per Figure 1) include a label with the game name at the top of the window. This class should be instantiated as `PokemonGame(master, grid_size=10, num_pokemon=15, task=TASK_ONE)`, where `TASK_ONE` is some constant (defined by you) that allows the game to be displayed as per Figure 1. While defaults are included for grid size and number of Pokemon, your game must still work as expected for other reasonable values of these parameters (`grid_size` $\in [2, 10]$ and `num_pokemon` $\in [0, \texttt{grid\_size}^2]$).

3.3 BoardView

`BoardView` represents the GUI for the board. At the beginning of the game the board should display all dark green (tall grass) squares. `BoardView` should inherit from `tk.Canvas` and should be instantiated as `BoardView(master, grid_size, board_width=600, *args, **kwargs)`, where '`*args, **kwargs`' signifies that you can include any additional arguments that you want. The `board_width` argument is the number of pixels the board should span (both width and height). The `grid_size` argument is the number of rows (equal to the number of columns) on the board. The grid does not need to be resizeable (i.e. the grid does not need to change size if the window is expanded), however, it must work for values of `grid_size` between 2 and 10. For task 1, you should use the `create_rectangle` method on the `BoardView` to construct and represent squares. When mouse click events occur on the `BoardView`, appropriate updates should occur (depending on the game play). Table 1 provides a summary of the effects that certain events should have. Note: On some operating systems, the tkinter event for a right click is `<Button-2>` and on others it is `<Button-3>`. To ensure this feature works for your marker, please bind the right click behaviour to **both** of these events.

A list of methods that may be useful to write in this class are included in Appendix A. You may also add your own methods where appropriate.

4 Task 2: Images, StatusBar, and File Menu - 6 marks

Task 2 requires you to add additional features to enhance the games look and functionality. Figure 2 and Figure 3 give examples of the game at the end of task 2. Note that unlike task 1, 0's do not need to be displayed in the squares for task 2. Other numbers, however, do need to be displayed. **Note: Your task 1 functionality must still be testable when the `task` parameter of `PokemonGame` is set to the `TASK_ONE` constant**. If you attempt task 2, you must define a `TASK_TWO` constant which, when supplied as the task parameter for `PokemonGame`, allows the app to run with any attempted task 2 features included.

4.1 StatusBar

Add a `StatusBar` class that inherits from `tk.Frame`. In this frame, you should include a game timer displaying the number of minutes and seconds the user has been playing the *current* game, as

Action	Game behaviour	Rectangle display
Left click on tall grass square with no hidden pokemon.	'Expose' tall grass to short grass.	Light green colour with superimposed text displaying the number of surrounding pokemon. If there are no surrounding pokemon, the number 0 should be displayed, and neighbouring cells should be exposed as per `big_fun_search` in the Assignment 1 support code.
Left click on tall grass square with hidden pokemon.	'Expose' all hidden pokemon, and provide a tkinter messagebox to tell the user they lost the game.	Yellow rectangles for squares that hide pokemon (including any previously caught pokemon).
Right click on unexposed square.	Toggle status (between 'attempted catch' and tall grass).	Red rectangle for 'attempted catch', dark green rectangle for tall grass.
Left click on an 'attempted catch' square.	No behaviour.	No change to game view.

Table 1: Board events and corresponding behaviours.

Figure 2: Example game at the end of task 2.

well as a representation of the number of current 'attempted catches', and the number of pokeballs the user has remaining. This information must be displayed alongside the relevant images (as per Figure 2). You must also include a 'New game' button and a 'Restart game' button, which allow the user to restart the current game. Both of these buttons must reset the information on the status bar, as well as setting all squares back to 'tall grass' squares. The 'New game' button should also generate new locations of hidden pokemon. **For full marks, the layout of the status bar must be as per Figure 2.**

4.2 End of game

When the player wins or loses the game, all pokemon should be exposed (pokemon images should be chosen at random), the game timer should be stopped, and the player should be informed of the outcome and prompted for whether to play again (see Figure 3). If they choose to play again, a new game should be prepared, and all game information should be reset (this must be communicated on the status bar). If they opt not to play again, the game should terminate.

4.3 Images

Create a new view class, `ImageBoardView` that *extends* your existing `BoardView` class. This class should behave similarly to the existing `BoardView` class, except that images should be used to display each square rather than rectangles (see the provided images folder). The view should be set up using the `ImageBoardView` when the game is run in `TASK_TWO` mode. You should still provide a functional `BoardView` class that allows us to test your task 1 functionality when `PokemonGame` is run in `TASK_ONE` mode.

4.4 File menu

Add a file menu with the options described in Table 2. Note that on Windows this will appear in the window, whereas on Mac this will appear at the top of your screen. For saving and loading files, you must design an appropriate file format to store information about games. You may use any format you like, as long as your save and load functionality work together.

Option	**Behaviour**
Save game	Prompt the user for the location to save their file (using an appropriate method of your choosing) and save all necessary information to replicate the current state of the game. Include appropriate error handling.
Load game	Prompt the user for the location of the file to load a game from and load the game described in that file. Include approriate error handling.
Restart game	Restart the current game, including game timer. Pokemon locations should persist.
New game	Restart to a new game (i.e. new pokemon locations). Use the same grid size and number of pokemon as the current game.
Quit	Prompt the player via messagebox to ask whether they are sure they would like to quit. If no, do nothing. If yes, quit the game (window should close and program should terminate).

Table 2: File menu options.

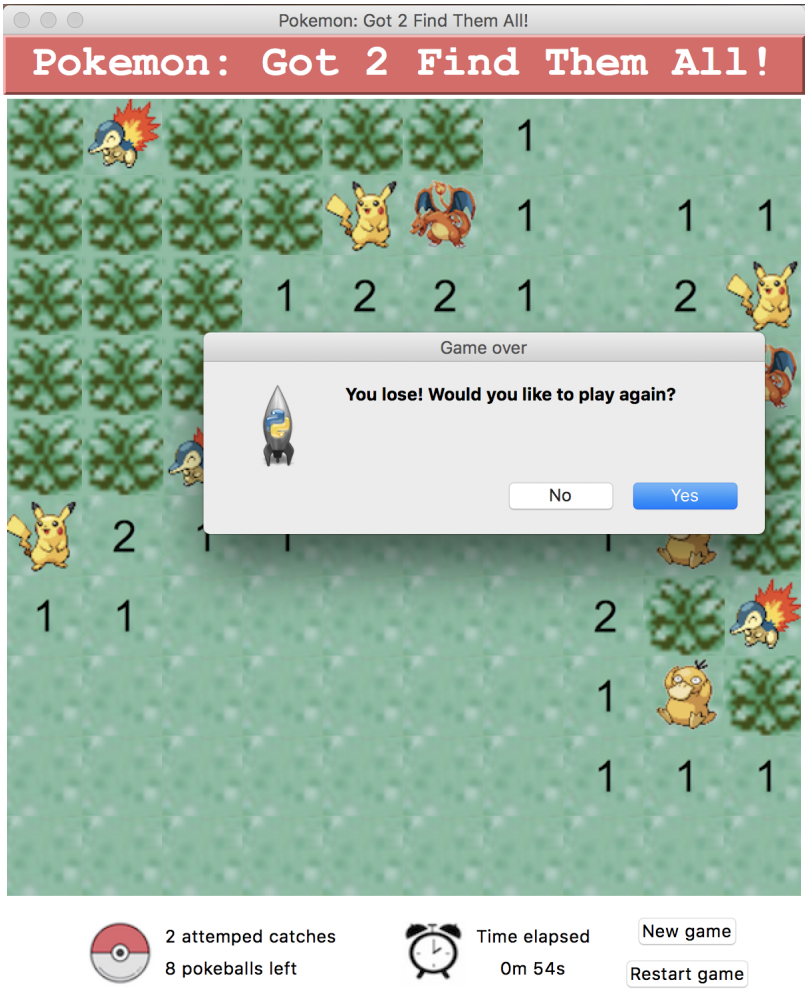

Figure 3: Game loss.

5 Postgraduate Task: Square highlighting and high scores - 5 marks

There are three additional tasks for postgraduate students. If you are enrolled in the undegraduate version of this course (CSSE1001), you may attempt these tasks, but you will not receive any marks for them.

5.1 Postgraduate task 1: High scores - 2 marks

To complete this task, you must add a 'High scores' option to the file menu you created in task 2. Selecting this option should create a top level window displaying an ordered leaderboard of the highest scores achieved by users in the game (up to the top 3); see Figure 5. The score is the users game time in seconds. These scores should persist even if the app is run again. When a user wins a game, you must prompt them for their name to display next to their score if they are within the top 3; see Figure 4. Integrate this feature into the displayed features when the game is run in `TASK_TWO` mode. You will likely need to write high score information to a file, and read from that file. You must ensure that if a file does not yet exist for these high scores, reading from and writing to such a file does not cause errors in your program. Requesting to see the leaderboard when no file exists yet should cause a window with only the 'High Scores' heading and 'Done' button to display. Entering a new high score when no file exists yet should cause a file to be created.

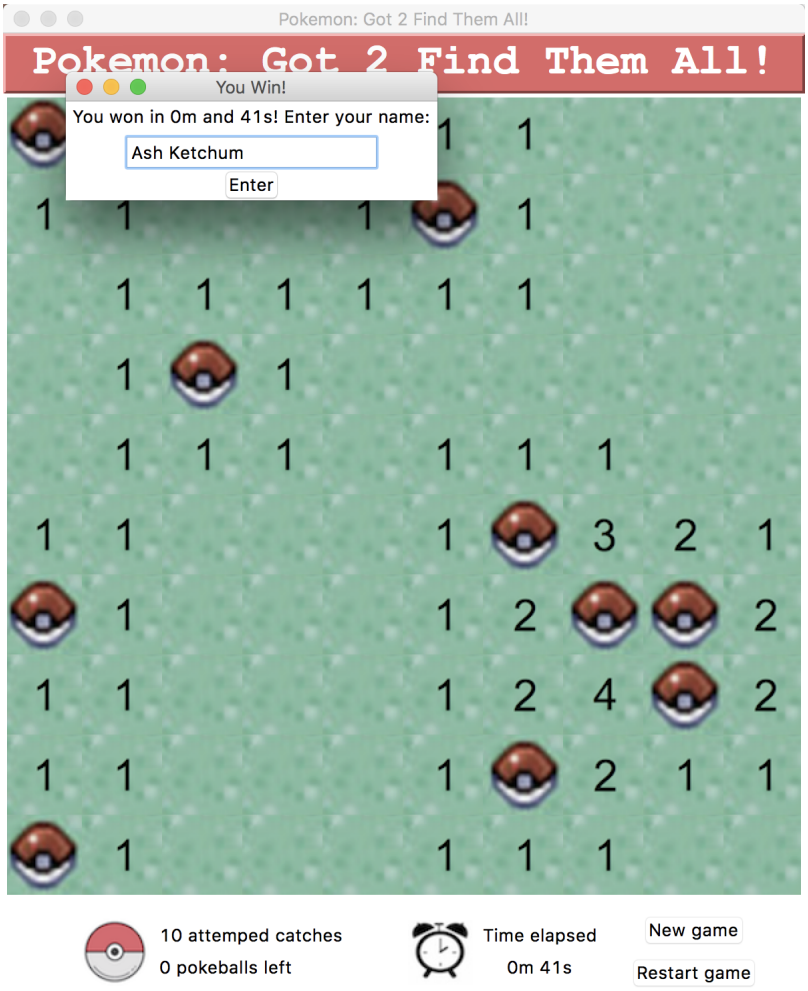

Figure 4: Prompt for name on game win.

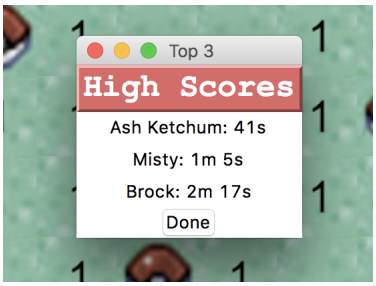

Figure 5: High score menu.

5.2 Postgraduate task 2: Square highlighting for Task 1 mode - 1 mark

When the game is run in `TASK_ONE` mode, motion onto a grid square should cause a border to appear around the square that the cursor is on. Motion off a grid square should cause this border to disappear. If you do not implement square highlighting for task 2 as per the next task, square highlighting should be disabled for task 2.

5.3 Postgraduate task 3: Square highlighting for Task 2 mode - 2 marks

When the game is run in `TASK_TWO` mode, motion on tall grass squares should cause the grass to 'rustle'. Motion onto a tall grass square should cause the image to change to the 'unexposed_moved.png' image, whereas motion off a tall grass square should restore the image to the 'unexposed.png' image.

6 Marking

Your total mark will be made up of functionality marks and style marks. Functionality marks are worth 16 of the 20 available marks for undergraduate students and 21 of the 25 available marks for postgraduate students. Style marks are worth the other 4 marks. The style of your assignment will be assessed by one of the tutors, and you will be marked according to the style rubric provided with the assignment. Your style mark will be calculated according to:

$$\text{Style} = 4 * (\frac{\text{Style Percentage}}{100} * \frac{\text{Task 1 Percentage}}{100})^{0.5}$$

Your assignment will be marked by tutors who will run your `a3.py` file and evaluate the completeness and correctness of the tasks you've implemented.

The table below specifies the mark breakdown for each of the tasks for CSSE1001 and CSSE7030 students.

Task	**CSSE1001 Marks**	**CSSE7030 Marks**
Task 1	10 marks	10 marks
Task 2	6 marks	6 marks
Postgraduate Task	0 marks	5 marks
Style	4 marks	4 marks
Total possible marks	20 marks	25 marks

Table 3: Mark breakdown for functionality.

7 Assignment Submission

Your assignment must be submitted via the assignment three submission link on Blackboard. You must submit **one** file, named `a3.py`. You do not need to resubmit any files supplied to you (e.g. the images).

Late submission of the assignment will **not** be accepted. In the event of exceptional circumstances, you may submit a request for an extension.

All requests for extension must be submitted on the UQ Application for Extension of Progressive Assessment form: https://my.uq.edu.au/node/218/2 **at least 48 hours prior** to the submission deadline. The application and supporting documentation must be submitted to the ITEE Coursework Studies office (78-425) or by email to enquiries@itee.uq.edu.au.

8 Appendices

8.1 Appendix A

This section outlines some methods that may be useful to write in the `BoardModel` and `BoardView` classes for task 1. Type hints and return types are omitted, as it is up to you to determine what these should be. Note that this list does not include `BoardModel` methods that may be described in Assignment 1, and is not necessarily complete. You may also need to add more methods to these classes for task 2 and/or the postgraduate task.

The word *index* is used to refer to the integer index into the game string, *position* is used to refer to the (row, col) coordinate, and *pixel* is used to refer to the (x, y) graphics coordinate described in some tkinter events.

8.1.1 BoardModel methods

Some methods that may be beneficial to write include:

- `get_game(self)`: Returns an appropriate representation of the current state of the game board.
- `get_pokemon_locations(self)`: Returns the indices describing all pokemon locations.
- `get_num_attempted_catches(self)`: Returns the number of pokeballs currently placed on the board.
- `get_num_pokemon(self)`: Returns the number of pokemon hidden in the game.
- `check_loss(self)`: Returns True iff the game has been lost, else False.
- `index_to_position(self, index)`: Returns the (row, col) coordinate corresponding to the supplied index.

8.1.2 BoardView methods

Some methods that may be beneficial to write include:

- `draw_board(self, board)`: Given an appropriate representation of the current state of the game board, draw the view to reflect this game state (note that if you are using this method every time an update occurs, you should first clear the entire board before drawing the game again).
- `get_bbox(self, pixel)`: Returns the bounding box for a cell centered at the provided pixel coordinates.
- `position_to_pixel(self, position)`: Returns the center pixel for the cell at position.
- `pixel_to_position(self, pixel)`: Converts the supplied pixel to the position of the cell it is contained within (pixel may not be the center pixel, it could be anywhere within the cell).