1 Introduction

In assignment 2 you implemented a game of *MazeRunner* with a text-based interface. The Apple MVC design pattern used in this game allows modelling logic and view classes to be modified fairly independently. In this assignment, you'll exchange the text-based interface of *MazeRunner* with a more sophisticated tkinter-based Graphical User Interface (GUI).

The game view initially contains 3 components:

- A level view on which the tiles and entities are drawn using either coloured shapes or images;
- An inventory view, which appears to the right of the level view and allows users to apply collected items (other than coins) by left-clicking those items in their inventory; and
- A stats view, which shows the player's stats, as well as the number coins they have collected.

As opposed to assignment 2, where users controlled the game by inputting text at a prompt, in assignment 3 users control the game through key-presses and mouse clicks. An example of the final game is shown in Figure 1.

You can find the tkinter documentation on effbot[1] and New Mexico Tech[2].

2 Tips and hints

This assignment is split into two tasks for undergraduate students, and an additional third task for postgraduate task. 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. You should be testing **regularly** throughout the coding process. Test your GUI manually and regularly upload to Gradescope to ensure the components you have implemented pass the Gradescope tests. At the minimum you should not move on to task 2 until you pass the task 1 tests.

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

[1] `https://web.archive.org/web/20171112065310/http://effbot.org/tkinterbook`
[2] `https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html`

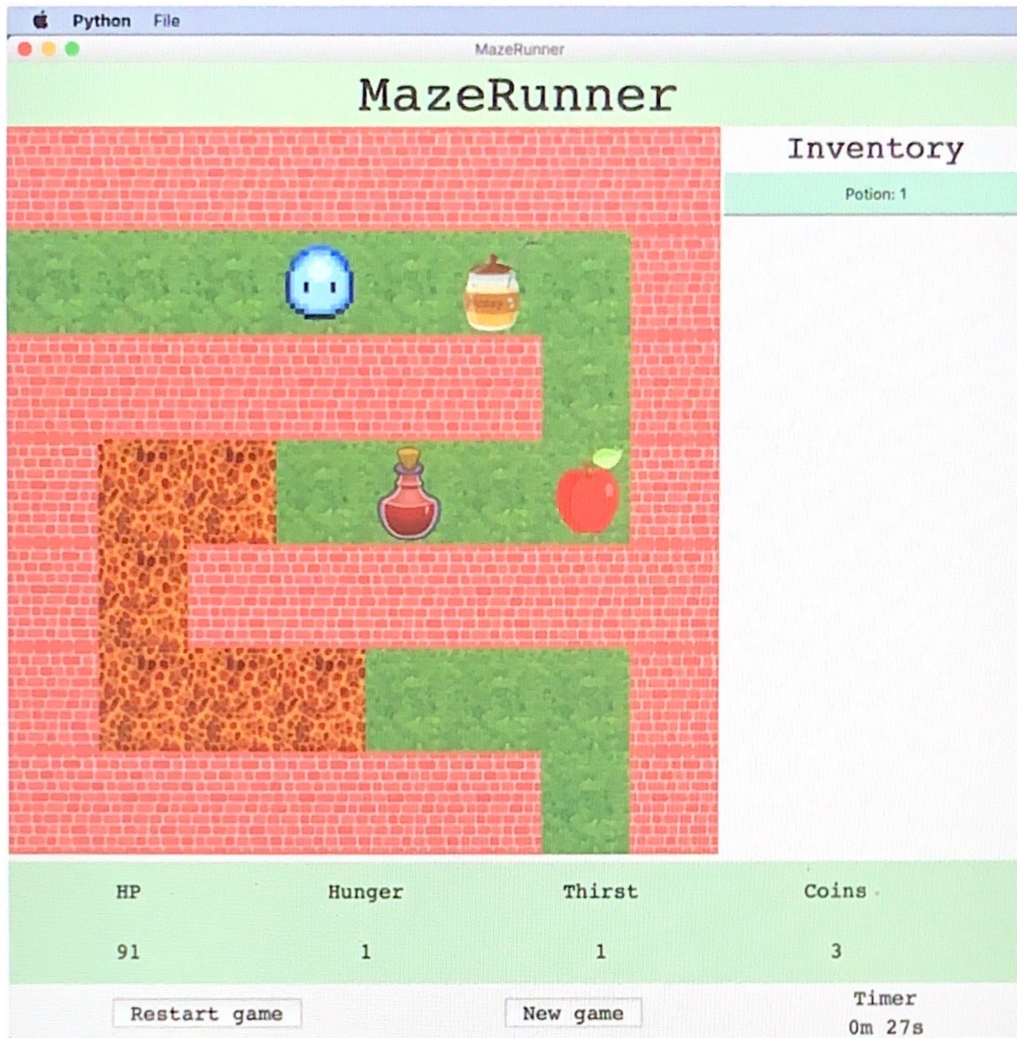

Figure 1: Example fully functional game at the end of Task 2.

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. If your solution contains code that prevents it from being run, you will receive a mark of 0.

You **must only** make use of libraries listed in Appendix A. You **must not** import anything that is not on this list; doing so will result in a deduction of **up to 100% of your mark**.

You may use any course provided code in your assignment. This includes any code from the support files or sample solutions for previous assignments **from this semester only**, as well as any lecture or tutorial code provided to you by course staff. However, it is your responsibility to ensure that this code is styled appropriately, and is an appropriate and correct approach to the problem you are addressing.

3 Task 1: Basic Gameplay - 10 marks

Task 1 requires you to implement a functional GUI-based version of *MazeRunner*. At the end of this task your game should look like Figure 2. A heading label should appear at the top of the window at all times. Below this, the three components should appear as described in Section 1 and as depicted below.

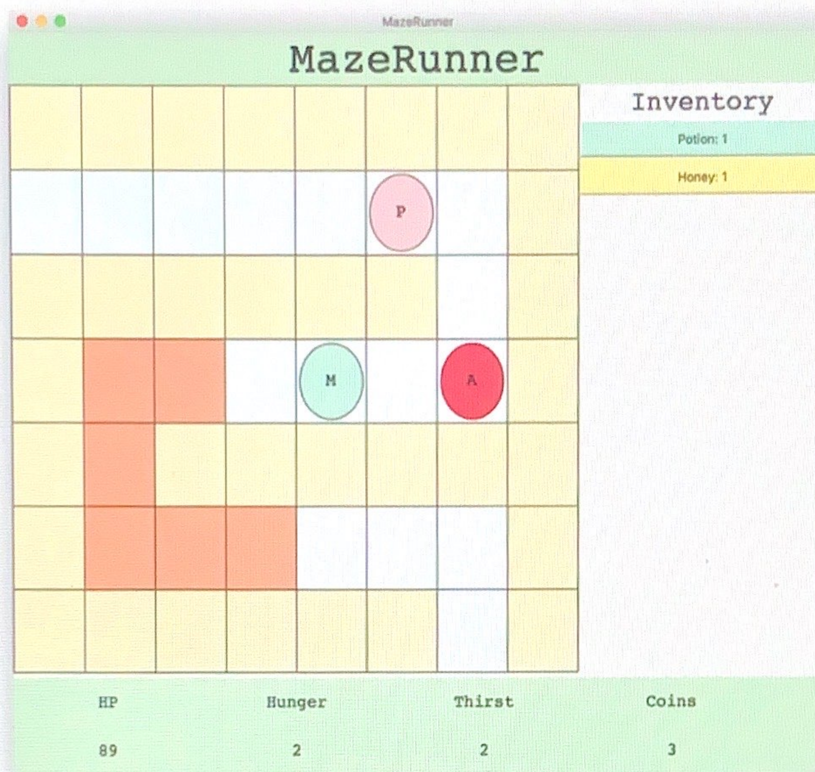

Figure 2: Game at end of task 1.

Certain events should cause behaviour as per Table 1.

Event	Behaviour
Key Press: 'w'	Player attempts to move up one square.
Key Press: 'a'	Player attempts to move left one square.
Key Press: 's'	Player attempts to move down one square.
Key Press: 'd'	Player attempts to move right one square.
Left click on an item in inventory	One item of the kind clicked should be applied to the player and removed from the inventory. If no instances of the item remain in the inventory, the label showing the item should be removed from view, and all other item labels below it should move up to fill the space.

Table 1: Events and their corresponding behaviours.

In task 1, tiles are represented by coloured rectangles and entities are represented by coloured circles. You must also annotate the circles of entities with their id (as per Fig. 2). The colours representing each tile and entity can be found in `constants.py`. You must use the colours specified in `constants.py`.

When the player wins or loses the game they should be informed of the outcome via a messagebox. The messagebox must display the text of `WIN_MESSAGE` or `LOSS_MESSAGE` in `constants.py`. For task 1, there is no required behaviour after the messagebox is closed; the program can terminate or remain open. However, whatever behaviour you choose must be reasonable (i.e. closing the messagebox should not cause your program to crash, or an error to show).

To complete this task you will need to implement various view classes, as well as a graphical controller class which extends the existing MazeRunner controller class. While it is not necessary in order to complete task 1, you are permitted to add additional modelling classes if they improve your solution.

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.

3.1 Provided Code

The following code is provided in `a2_solution.py` and `a3_support.py` for you to use when implementing your solution.

3.1.1 Model classes

Model classes should be defined as per Assignment 2. You may add more modelling classes *if they improve the code*. You may **not** modify the supplied model classes.

3.1.2 AbstractGrid

`AbstractGrid` is an abstract view class which inherits from `tk.Canvas` and provides base functionality for multiple view classes. An AbstractGrid can be thought of as a grid with a set number of rows and columns, which supports creation of text and shapes at specific (row, column) position. Note that the number of rows may differ from the number of columns, and may change after the construction of the AbstractGrid. If the dimensions change, the size of the cells will be be updated to allow the AbstractGrid to retain its original size. AbstractGrid consists of the following interface:

- `__init__(self, master: Union[tk.Tk, tk.Frame], dimensions: tuple[int, int], size: tuple[int, int], **kwargs) -> None`:
  Sets up a new `AbstractGrid` in the `master` frame. `dimensions` is the initial number of rows and number of columns, and `size` is the width in pixels, and the height in pixels. `**kwargs` is used to allow `AbstractGrid` to support any named arguments supported by `tk.Canvas`.
- `set_dimensions(self, dimensions: tuple[int, int]) -> None`:
  Sets the dimensions of the grid to `dimensions`.
- `get_bbox(self, position: tuple[int, int]) -> tuple[int, int, int, int]`:
  Returns the bounding box for the (row, col) position, in the form (x_min, y_min, x_max, y_max).
- `get_midpoint(self, position: tuple[int, int]) -> tuple[int, int]`:
  Gets the graphics coordinates for the center of the cell at the given (row, col) position.

- `annotate_position(self, position: tuple[int, int], text: str) -> None`:
  Annotates the cell at the given (row, col) position with the provided text.
- `clear(self) -> None`: Clears the canvas.

3.2 View classes

You must implement the view classes for the level map, inventory, and player stats. Because the level map and player stats can both be represented by grids, you are required to use the AbstractGrid class to factor out the shared functionality. This section outlines the classes and methods you are *required* to write. While you do not need to, you are permitted to add additional methods or classes provided they improve your solution.

3.2.1 LevelView

`LevelView` is a view class which inherits from `AbstractGrid` and displays the maze (tiles) along with the entities. Tiles are drawn on the map using coloured rectangles at their (row, column) postitions, and entities are drawn over the tiles using coloured, annotated ovals at their (row, column) positions (as per Fig. 2). The colours representing each tile and entity can be found in `constants.py`. You must use these colours.

Your program should work for reasonable map sizes. You must not assume that the number of rows will always be equal to the number of columns. You must use the `create_oval`, `create_rectangle`, and `create_text` methods for `tk.Canvas` to achieve this task. The `LevelView` class should be instantiated as `LevelView(master, dimensions, size, **kwargs)` where `**kwargs` should be passed to the super class as with AbstractGrid. The width and height of the maze shown in the level (and consequently, the level view) are given in `constants.py`. While it is recommended to create your own helper methods in this class, the only method you are required to create is:

- `draw(self, tiles: list[list[Tile]], items: dict[tuple[int, int], Item], player_pos: tuple[int, int]) -> None`:
  Clears and redraws the entire level (maze and entities).

3.2.2 StatsView

`StatsView` is a view class which inherits from `AbstractGrid` and displays the player's stats (HP, health, thirst), along with the number of coins collected (as per Fig. 2). A `StatsView` always has four columns and two rows. The first row contains the headings for the types of stats and the second row contains the values for the stat in that column. You are required to implement the following methods in this class:

- `__init__(self, master: Union[tk.Tk, tk.Frame], width: int, **kwargs) -> None`:
  Sets up a new `StatsView` in the `master` frame with the given `width`. The height of the `StatsView` can be found in `constants.py`. The background colour should be set to `THEME_COLOUR` from `constants.py` via the `**kwargs`.
- `draw_stats(self, player_stats: tuple[int, int, int]) -> None`:
  Draws the player's stats (hp, hunger, thirst).
- `draw_coins(self, num_coins: int) -> None`: Draws the number of coins.

Again, it may be beneficial to create your own helper methods in this class.

3.2.3 InventoryView

`InventoryView` is a view class which inherits from `tk.Frame`, and displays the items the player has in their inventory via `tk.Labels`, under a title label. This class also provides a mechanism through which the user can apply items. You must implement the following methods in this class:

- `__init__(self, master: Union[tk.Tk, tk.Frame], **kwargs) -> None`:
  Creates a new `InventoryView` within `master`.
- `set_click_callback(self, callback: Callable[[str], None]) -> None`:
  Sets the function to be called when an item is clicked. The provided `callback` function should take one argument; the string name of the item.
- `clear(self) -> None`: Clears all child widgets from this `InventoryView`.
- `_draw_item(self, name: str, num: int, colour: str) -> None`:
  Creates and binds (if a callback exists) a single `tk.Label` in the `InventoryView` frame. `name` is the name of the item, `num` is the quantity currently in the users inventory, and `colour` is the background colour for this item label (determined by the type of item).
- `draw_inventory(self, inventory: Inventory) -> None`:
  Draws any non-coin inventory items with their quantities and binds the callback for each, if a click callback has been set. Hint: loop over each item in the inventory and call `_draw_item` for each to create and bind the item label.

3.2.4 GraphicalInterface

`GraphicalInterface` inherits from `UserInterface` and must implement the methods described by `UserInterface`. The `GraphicalInterface` manages the overall view (i.e. the title banner and the three major widgets), and enables event handling. You must implement the following methods:

- `__init__(self, master: tk.Tk) -> None`: Creates a new `GraphicalInterface` with master frame `master`. Sets the title of the window to `'MazeRunner'` and creates a title label. Note: this method is not responsible for instantiating the three major components, as you may not know the dimensions of the maze when the `GraphicalInterface` is instantiated.
- `create_interface(self, dimensions: tuple[int, int]) -> None`:
  Creates the components (level view, inventory view, and stats view) in the master frame for this interface. `dimensions` represent the (row, column) dimensions of the maze in the current level.
- `clear_all(self) -> None`: Clears each of the three major components (do not delete the component instances, just clear them).
- `set_maze_dimensions(self, dimensions: tuple[int, int]) -> None`:
  Updates the dimensions of the maze in the level to `dimensions`.
- `bind_keypress(self, command: Callable[[tk.Event], None]) -> None`:
  Binds the given `command` to the general keypress event. The command should be a function which takes in the keypress event, and performs different actions depending on what character was pressed. Any character other than 'w', 'a', 's', or 'd' should be ignored (including all uppercase letters).

- `set_inventory_callback(self, callback: Callable[[str], None]) -> None`:
  Sets the function to be called when an item is clicked in the inventory view to be `callback`. The `callback` function will need to be constructed in the controller class (see Section 3.3), and must take one argument (the string name of an Item), and apply the first instance of that item in the inventory to the player. This function should then remove that item from the player's inventory.

- `draw_inventory(self, inventory: Inventory) -> None`:
  Draws any non-coin inventory items with their quantities and binds the callback for each, if a click callback has been set.

- `draw(self, maze: Maze, items: dict[tuple[int, int], Item], player_position: tuple[int, int], inventory: Inventory, player_stats: tuple[int, int, int]) -> None`: Must implement the `draw` method as per the docstring in the `UserInterface` class. This should just involve clearing the three major components and redrawing them with the new state (provided as arguments to this method).

- `_draw_inventory(self, inventory: Inventory) -> None`:
  Must implement the `_draw_inventory` method as per the docstring in the `UserInterface` class. Note: this method will need to draw both the non-coin items on the inventory view (see public `draw_inventory` method), and also draw the coins on the stats view.

- `_draw_level(self, maze: Maze, items: dict[tuple[int, int], Item], player_position: tuple[int, int]) -> None`:
  Must implement the `_draw_level` method as per the docstring in the `UserInterface` class.

- `_draw_player_stats(self, player_stats: tuple[int, int, int]) -> None`:
  Must implement the `_draw_player_stats` method as per the docstring in the `UserInterface` class.

3.3 Controller class

3.3.1 GraphicalMazeRunner

`GraphicalMazeRunner` should inherit from `MazeRunner` and overwrite / add methods where required in order to enable the game to use a `GraphicalInterface` instead of a `TextInterface`, and to be based on events generated by the user (keypresses and mouse clicks), rather than based on user input to the shell. In particular, you will need to implement the following methods:

- `__init__(self, game_file: str, root: tk.Tk) -> None`: Creates a new GraphicalMazeRunner game, with the view inside the given root widget.

- `_handle_keypress(self, e: tk.Event) -> None`: Handles a keypress. If the key pressed was one of 'w', 'a', 's', or 'd' a move is attempted. If the player wins or loses the game with this move, they are informed of their result via a messagebox.

- `_apply_item(self, item_name: str) -> None`: Attempts to apply an item with the given name to the player.

- `play(self) -> None`: Called to cause gameplay to occur. This method should first create the widgets on the `GraphicalInterface`, bind the keypress handler, set the inventory callback, and then redraw to allow the game to begin.

3.4 play_game(root: tk.Tk) function

The play_game function should be fairly short. You should:

1. Construct the controller instance using the file in the GAME_FILE constant and the root tk.Tk parameter.
2. Cause gameplay to commence.
3. Ensure the root window stays opening listening for events (using mainloop).

3.5 main function

The main function should:

1. Construct the root tk.Tk instance.
2. Call the play_game function passing in the newly created root tk.Tk instance.

4 Task 2: Images, Buttons, and File Menu

Task 2 requires you to add additional features to enhance the games look and functionality. Figure 1 gives an example of the game at the end of task 2. Another example is shown below in Figure 3. **Note: Your task 1 functionality must still be testable. When your program is run with the TASK constant in constants.py set to 1, the game should display only task 1 features. When your program is run with the TASK constant set to 2, the game should display all attempted task 2 features.** There should be no task 2 features visible when running the game in task 1 mode.

As an advanced task, the structure of your code is largely up to you. However, if you complete this task, you will be marked on how well-designed your code is. Some aspects of design are compulsory (e.g. implementing the subclasses described in subsections below). However, the methods and internal design of these classes are not prescribed. When designing your solution, you should consider ways to effectively utilize the inheritance structure to avoid duplicating code.

4.1 Images

Create a new view class, ImageLevelView, that *extends* your existing LevelView class. This class should behave similarly to the existing LevelView class, except that images should be used to display the tiles and entities, rather than rectangles and ovals (see the provided images folder). When your assignment is run with the TASK constant in constants.py set to 2, the ImageLevelView should be displayed. When it is run with the TASK constant in constants.py set to 1, the basic LevelView from task 1 should be displayed.

Just as the rectangles and ovals needed to change size depending on the dimensions of the level maze in task 1, in task 2, the images must resize appropriately. In order to do this, you will need to install Pillow[3], and then import ImageTk and Image from PIL.

[3] https://pillow.readthedocs.io/en/stable/installation.html

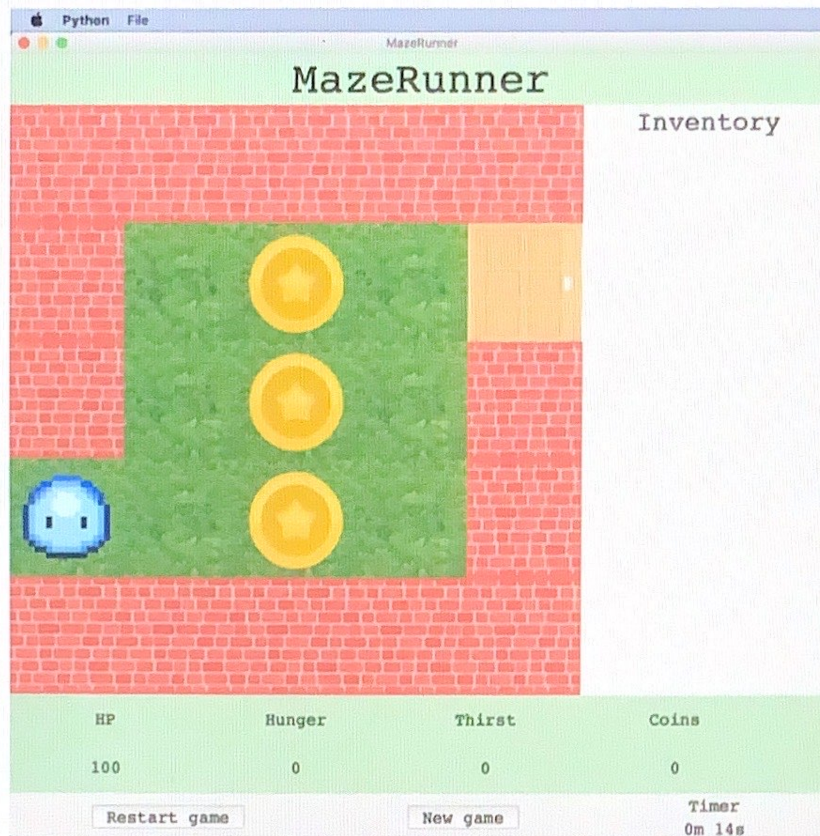

Figure 3: Another example fully functional game at the end of Task 2.

4.2 ControlsFrame

Add a new class `ControlsFrame` which inherits from `tk.Frame`, and displays two buttons (restart and new game), as well as a timer of how long the current game has been going for. The `ControlsFrame` instance should be displayed at the bottom of the interface, just below the `StatsView`. The details of the three components on this widget are as follows:

- Restart button: when this button is clicked, the current game should be restarted, i.e. the user should return to the start of the first level, their move count and stats should reset, and the game timer should return to 0.
- New game button: when this button is clicked, a `tk.TopLevel` window should appear, prompting the user to enter a relative path to a new game file. If the user enters a valid game file, the game should restart using this game file. If they enter an invalid game file, they should be informed that the game file was not valid via a messagebox. Once the user has acknowledged the messagebox, the toplevel window should close and the user should *not* be automatically reprompted.
- The game timer should display the number of minutes and seconds that have elapsed since the current game began.

4.3 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. On Mac, the file menu should look something like Figure 4. The exact layout is not overly important (e.g. your menu

may have a dashed line at the top, or other minor differences in display), as long as the options are present with the correct text, and correct functionality.

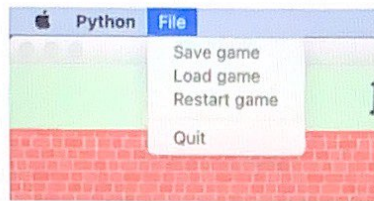

Figure 4: Example file menu on Mac.

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 you do not import anything that has not been explicitly permitted in Appendix A, and your save and load functionality work together. Reasonable actions by the user (e.g. trying to load a non-game file) should be handled appropriately (e.g. with a pop-up to inform the user that something went wrong).

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.
Load game	Prompt the user for the location of the file to load a game from and load the game described in that file.
Restart game	Restart the current game, including game timer.
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.