CSSE1001 Assignment 3

Due 5pm, Friday 30th October 2020

1 Introduction

In Assignment 2 you implemented a text-based version of the *Key Cave Adventure Game*. In Assignment 3 you will modify this game to a graphical user interface (GUI) based game. Your implementation should maintain the Apple model-view-controller (Apple MVC) structure used in Assignment 2. In this assignment, the View must be implemented using tkinter.

The new version of this game is a single-player GUI-based game in which the player is presented with a grid of squares (represented by either coloured rectangles or images). The objective is for the ibis (the player) to collect the trash and take it to their nest. This game is full of adventure. The player can move either by key presses or by clicking on an on-screen keypad.

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 **must only** make use of libraries listed in Appendix A. You must not import anything that is not on this list.

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.

You must write all your code in one file, titled `a3.py`. You must only submit this file; you are not permitted to submit any other files. Your game must display (in the latest attempted mode) when your marker runs this file. This means that if you *import* code from previous assignments or previous support files, your code will not run when we try to run it. If you would like to use course provided code, this code must be present within your `a3.py` file.

3 Task 1: Basic Gameplay - 10 marks

Task 1 requires you to implement a functional GUI-based version of game of *Key Cave Adventure Game*. To do this you will need to implement some View classes and make appropriate modifications to the Controller class from Assignment 2.

Certain events should cause behaviour as per Table 1.

Event	Behaviour
Key Press: ‘w’	User moves up one square if the map permits.
Key Press: ‘a’	User moves left one square if the map permits.
Key Press: ‘s’	User moves down one square if the map permits.
Key Press: ‘d’	User moves right one square if the map permits.
Left click on keypad	User moves in the direction shown on the rectangle they click. If they click on the keypad at a place that is not displaying a direction, nothing should happen.

Table 1: Events and their corresponding behaviours.

When the player wins or loses the game they should be informed of the outcome via a messagebox. After the user closes the messagebox the game may terminate, or may remain open (further actions will not be tested; i.e. your tutor will not try to continue playing the game).

In task 1, entities are represented by coloured rectangles. You must also annotate the rectangles of certain entities with what they represent (as per Fig. 1). The colours representing each entity are:

- Wall: Dark grey
- Ibis (Player): Medium spring green
- Banana (MoveIncrease): Orange
- Trash (Key): Yellow
- Nest (Door): Red

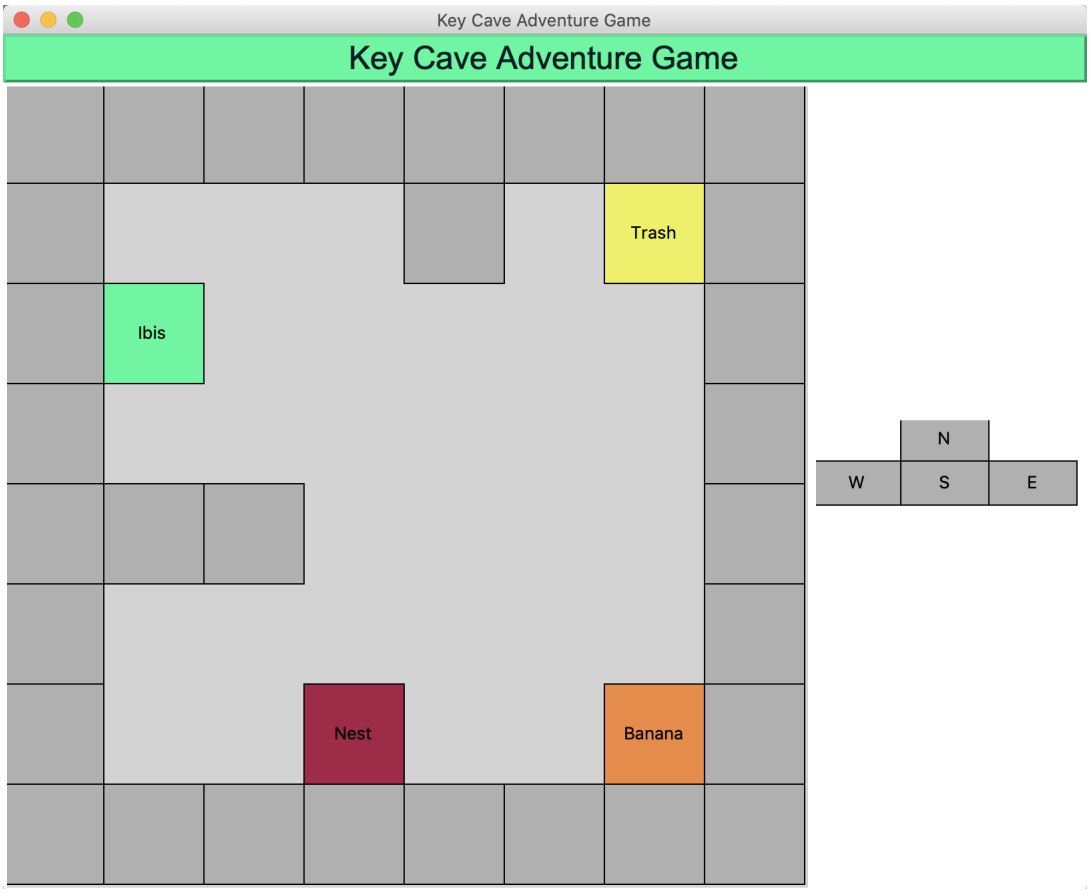

Figure 1: Example game at the end of task 1.

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 Model classes

Model classes should be defined as per Assignment 2. You may add more modelling classes if they improve the code. You should at least include `Entity`, `Wall`, `Player`, `Item`, `Key`, `MoveIncrease`, `Door`, and `GameLogic`. Note that, though we're now representing these classes with different concepts (e.g. the `Door` will be represented with a graphical depiction of a nest) you don't necessarily need to modify these classes to modify how they're represented in the graphical interface. You may choose to modify the names of the classes if this makes it easier for you to understand, but you won't need to.

3.2 View Classes

You must implement view classes for the game map and the keypad. However, because these can both be represented by grids of rectangles and share a fair amount of functionality, you are required to create an abstract class to factor out this shared functionality. Appendix B outlines methods that may be useful to write in each of these classes. You may add additional methods if they improve the design of your classes.

3.2.1 AbstractGrid

`AbstractGrid` is an abstract view class which inherits from `tk.Canvas` and provides base functionality for many of the view classes, as they share a fair amount of common functionality. Namely, they can be thought of as a grid with a set number of rows and columns, and they should all support creation of text at specific positions based on row and column. Though `AbstractGrid` is an abstract class, if it were to be instantiated, it would be as `AbstractGrid(master, rows, cols, width, height, **kwargs)` where '`**kwargs`' signifies that any named arguments supported by `tk.Canvas` class should also be supported by `AbstractGrid`. Note that the number of rows may differ from the number of columns.

3.2.2 DungeonMap

`DungeonMap` is a view class which inherits from `AbstractGrid`. Entities are drawn on the map using coloured rectangles at different (row, column) positions. You may assume that the number of rows will always be the same as the number of columns for the `DungeonMap`; i.e. the map is always a square. Your program should work for reasonable dungeon sizes (#rows $\in [4, 15]$). Some entities have their IDs written on their cells (see Figure 1). You must use the `create_rectangle` and `create_text` methods for `tk.Canvas` to achieve this. You should set the background colour of the `DungeonMap` instance to light gray by using the `kwargs`. The `DungeonMap` class should be instantiated as `DungeonMap(master, size, width=600, **kwargs)`, where `size` is the number of rows (equal to the number of columns) in the grid, and `width` is the number of pixels for the width and height of the grid.

3.2.3 KeyPad

`KeyPad` is a view class which inherits from `AbstractGrid`, and represents the GUI keypad. Each key is represented as a rectangle created on the `KeyPad` canvas itself, with text superimposed over it (also created on the `KeyPad` canvas itself). You must use the `create_rectangle` and `create_text` methods for `tk.Canvas` to achieve this. The `KeyPad` class should be instantiated as `KeyPad(master, width=200, height=100, **kwargs)`.

3.3 GameApp

`GameApp` 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 `GameApp(master, task=TASK_ONE, dungeon_name="game2.txt")`, where `TASK_ONE` is some constant (defined by you) that allows the game to be displayed as per Figure 1. The `dungeon_name` is the name of the file to load the level from. Though this argument has a default value, all attempted features should work with other reasonable game files.

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 gives an example of the game at the end of task 2. **Note: Your task 1 functionality must still be testable when the `task` parameter of `GameApp` 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 `GameApp`, allows the app to run with any attempted task 2 features included. There should be no task 2 features visible when running the game in `TASK_ONE` mode.

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

4.1 StatusBar - 1.5 marks

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 alongside the clock image, as well as a counter of the number of moves remaining alongside the lightning image. You must also include a 'Quit' button (which ends the program), and a 'New game' button, which allows the user to start the game again (this must reset the information on the status bar, as well as setting the map back to how it appears at the start of a game). **For full marks, the layout of the status bar must be as per Figure 2.**

4.2 End of game - 0.5 marks

When the player wins or loses the game the game timer should be stopped and the player should be informed of the outcome (including their score if they won) and prompted for whether to play again (see Figure 3). The score is the user's game time (lower is better). 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 - 2 marks

Create a new view class, `AdvancedDungeonMap` that *extends* your existing `DungeonMap` class. This class should behave similarly to the existing `DungeonMap` 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 `AdvancedDungeonMap` when the game is run in `TASK_TWO` mode. You should still provide a functional `DungeonMap` class that allows us to test your task 1 functionality when `GameApp` is run in `TASK_ONE` mode. You will need to resize the images appropriately depending on the level file. You may find the Pillow library useful for this task.

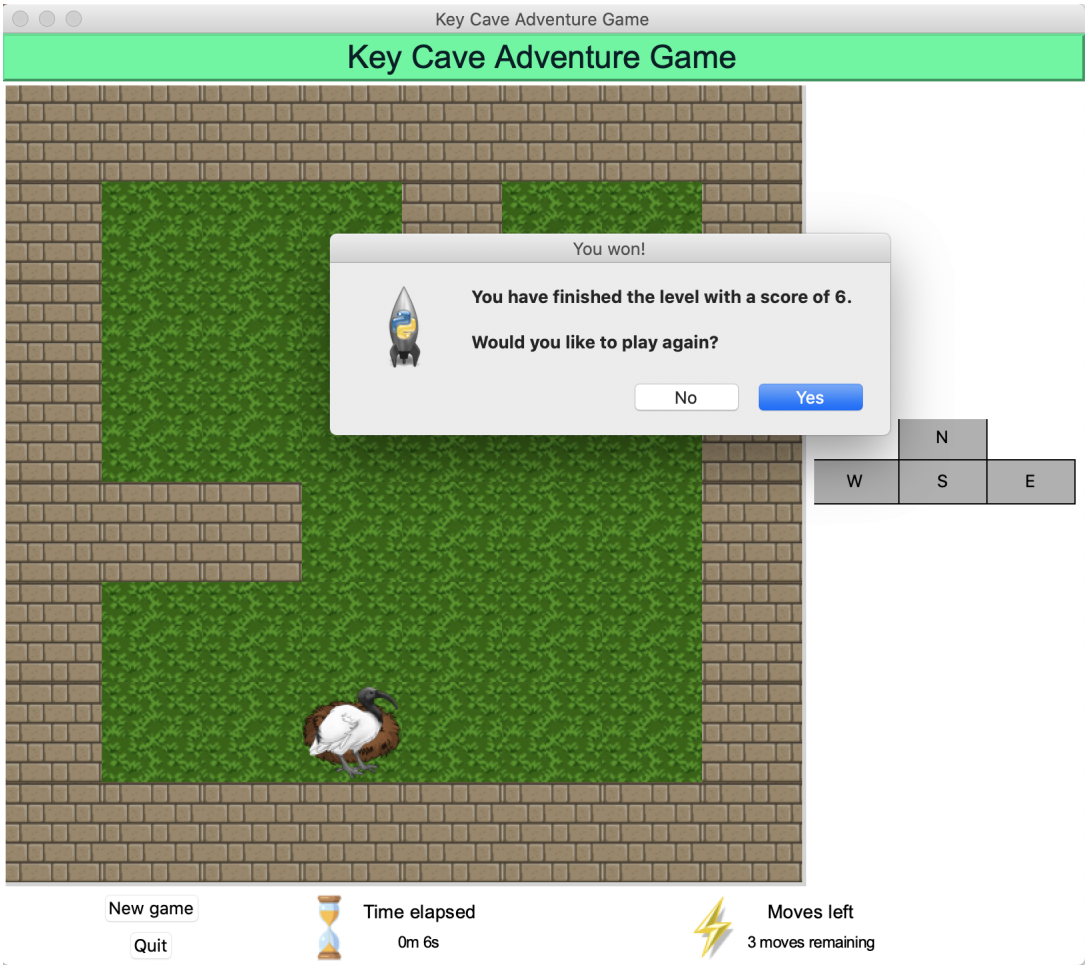

Figure 3: Game win (task 2).

4.4 File menu - 2 marks

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. 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.
New 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.

5 Postgraduate Task: High scores and lives - 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. You must create a `MASTERS` constant, which can be passed to the `GameApp` instance as the `task` argument to run the game with any

attempted postgraduate features visible. No postgraduate task features should be visible when running the game in `TASK_ONE` or `TASK_TWO` mode.

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 best (i.e. lowest) scores achieved by users in the game (up to the top 3); see Figure 5. 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. 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.

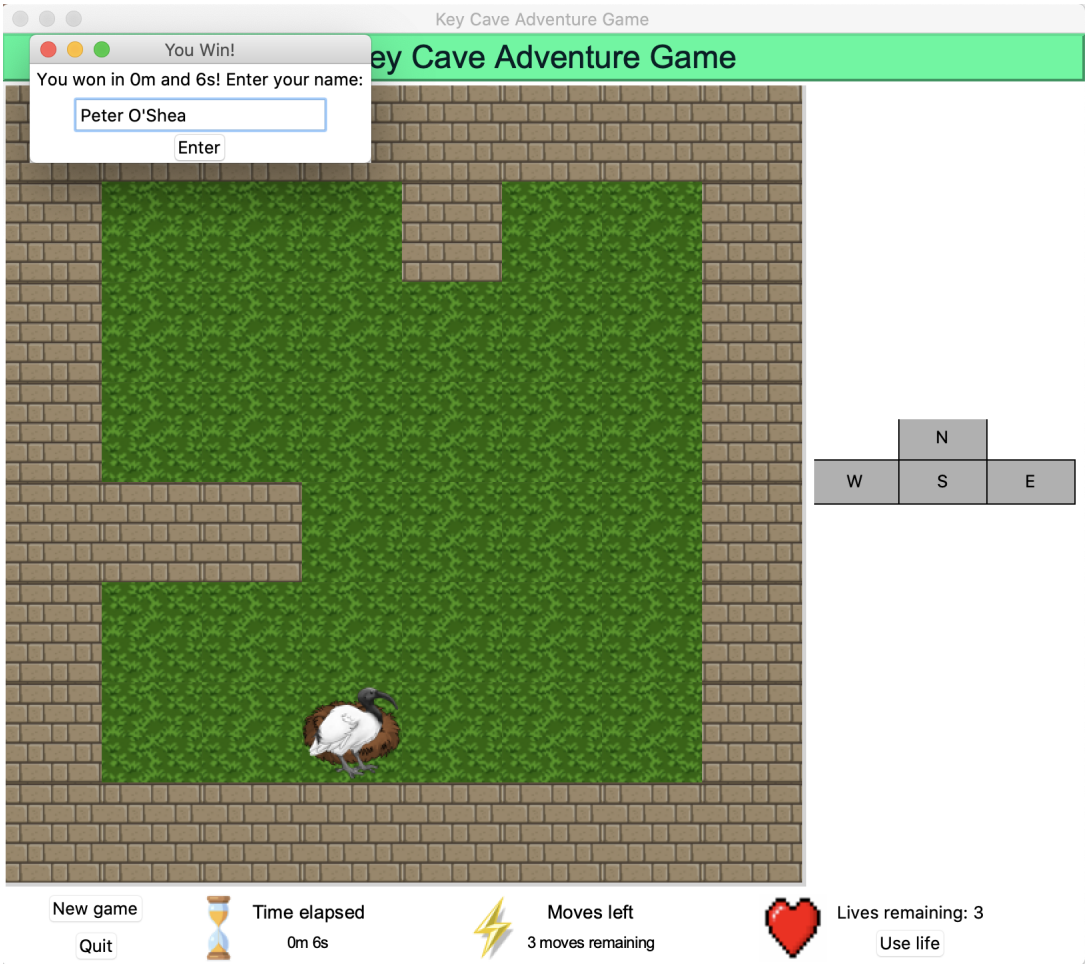

Figure 4: Prompt for name on game win.

Figure 5: High score menu.

5.2 Postgraduate task 3: Lives - 3 marks

The player should start any new game with 3 'lives'. Each 'life' gives the user a chance to undo the most recent move. When the user uses a life, the move count should be updated accordingly, their game time should be set back to what it was immediately before they made their last move, and one 'life' should be removed from their remaining lives. The number of lives remaining should be communicated in the status bar alongside the 'lives.png' image (as per Figure 4). You must not include this feature on the standard `StatusBar`. Instead, you should create a new subclass of the `StatusBar` class for this task, which is only displayed when the `task` constant is set to `MASTERS`.

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:

$$\text{Final style mark} = \text{Style rubric mark} * \min(10, \text{Functionality mark})/10$$

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: Permitted libraries

You may import the following libraries if you wish:

1. tkinter
2. PIL

8.2 Appendix B: Suggested methods for Task 1 classes

This section outlines some methods that may be useful to write in the view 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 is not necessarily complete. You may also need to add more methods to these classes for task 2 and/or the postgraduate task. You may choose not to implement some of these methods, but your code must still be well-designed.

8.2.1 AbstractGrid

The following methods may be useful to include in the `AbstractGrid` class.

- `get_bbox(self, position)`: Returns the bounding box for the (row, col) position.
- `pixel_to_position(self, pixel)`: Converts the x, y pixel position (in graphics units) to a (row, col) position.
- `get_position_center(self, position)`: Gets the graphics coordinates for the center of the cell at the given (row, col) position.
- `annotate_position(self, position, text)`: Annotates the cell at the given (row, col) position with the provided text.

8.2.2 DungeonMap

- `draw_grid(self, dungeon, player_position)`: Draws the dungeon on the `DungeonMap` based on `dungeon`, and draws the player at the specified (row, col) position.

8.2.3 KeyPad

- `pixel_to_direction(self, pixel)`: Converts the x, y pixel position to the direction of the arrow depicted at that position.