Extra Fancy Sokoban

1 Introduction

In A 2 you implemented a text-based game of *Fancy Sokoban* using the Model-View-Controller design pattern. In A 3 , you will swap out the text-based interface for a graphical user interface (GUI) using tkinter. An example of a final completed game is displayed in Figure 1.

Figure 1: Example screenshot from a completed *Extra Fancy Sokoban* implementation. Note that your display may look slightly different depending on your OS.

As opposed to earlier assignments where user interaction was handled via calls to `input`, user interaction in A 3 will occur via events such as key-presses and mouse clicks.

Your solution will still need to follow the Apple MVC design pattern covered in lectures. Because we have followed the MVC pattern, we can reuse the modelling classes from A 2 for this graphical implementation. These modelling classes (with some enhanced capabilities) have been provided for you. In addition to these modelling classes, some extra support code and constants have been provided to support you in your assignment; see Section 4 for further details. You are required to implement [illegible] classes as well as the controller class.

2 Setting Up

Aside from downloadin[illegible]3.zip, to begin this assignment you will also need to install the Pillow librar[illegible]ons on how to install Pillow can be found here[1].

3 Tips and hints

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. Note that Gradescope tests may fail on an implementation that visually appears correct if your implementation is wrong. You must implement your solution according to the implementation details from Section 6. Implementing the game using your own structure is likely to result in a grade of 0. Note also that minor differences in your program (e.g. a few pixels difference in widget size) may not cause the tests to fail. It is your responsibility to upload to Gradescope early and often, in order to ensure your solution passes the tests.

This document outlines the required classes and methods in your assignment. You are *highly encouraged* to create your own helper methods to reduce code duplication and to make your code more readable.

You **must not add any imports**; doing so will result in a deduction of **up to 100% of your mark**.

For additional help with tkinter, you can find documentation on effbot[2] and New Mexico Tech[3].

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

4 Provided Code

This section provides a brief, high-level overview of the files provided for you in `a3.zip`. For further information, please see the documentation within each file.

4.1 a2_support.

This file is the support o you in A 2.

4.2 model.py

The `model.py` file provi es for the Sokoban game. It is almost an exact solution to A 2, but with some extensions. For example, this model includes coins and supports a basic s hop for buying potions with those coins. It also provides some additional methods that may be useful in later parts of A3, such as support for resetting a game. You should only need to instanti ate, and retain a reference to, a `SokobanModel` instance in your code. However, you will still ne ed to interact with instances of the other classes via the `SokobanModel` instance.

4.3 a3_support.py

The `a3_support.py` file contains support code to assist you in writing your solution. In particular, this file provides the following:

1. A number of useful constants that you should use within your solution.

2. `get_image(image_name: str, size: tuple[int, int], cache: dict[str, ImageTk.PhotoImage] = None) -> Image`: a function to create, resize, and optionally cache images based on the name of their image file. Returns the image object, which can be rendered onto a tkinter Canvas. ***Note:* you need to retain references to either all the images, or to the cache.** Tkinter will delete an image as soon as all references to it have been lost. ***Note also:* use of this function in creating images is mandatory.**

3. `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) positions. Note that the number of rows may differ from the number of columns, and may change after the construction of the AbstractGrid.

4.4 maze_files/

This is a folder containing some example maze files which you can use for testing. You should also consider creating your own maze files to test edge cases.

4.5 images/

This is a folder containing images to use within your assignment.

5 Recommended Approach

As opposed to earlier assignments, where you would work through the task sheet in order, developing GUI programs tends to require that you work on various interacting classes in parallel. Rather

than working on each class in the order listed, you may find it beneficial to work on one *feature* at a time and test it thoroughly before moving on. Each feature will require updates / extensions to the controller, and potentially additions to one or more view classes. The recommended order of features (after reading through all of Section 6 of this document) are as follows:

1. `play_game`, `main`, and title: Create the window, ensure it displays when the program is run and set its title. (You must implement `play_game` in order to test your code, so you cannot earn marks until you have implemented this function.
2. Title banner: Render the banner at the top of the window.
3. `FancyGameView`:
   - Basic tile display
   - Entities (incl. player) display on top of tiles. Annotating strength value on crates.
   - Player movement
   - Player win / loss
4. `FancyStatsView`:
   - Basic display (non-functional). This step could also be done before the `FancyGameView`.
   - Functionality (ability to update).
5. `Shop`
   - Basic display
   - Handling buying items

6 Implementation

You must implement three view components; `FancyGameView`, `FancyStatsView`, and `Shop`. You must also implement a `FancySokobanView` class, which represents the overall view, and constructs and manages these smaller components. Additionally, you must implement a controller class - `ExtraFancySokoban` - which instantiates the `SokobanModel` and the `FancySokobanView` classes, and handles events and facilitates communication between the model and view classes.

This section describes the required structure of your implementation, however, it is not intended to provide an order in which you should approach the tasks. The controller class will likely need to be implemented in parallel with the view classes. See Section 5 for a recommended order in which you should approach this assignment.

6.1 FancyGameView

`FancyGameView` should inherit from `AbstractGrid` (see `a3_support.py`). The `FancyGameView` is a grid displaying the game map (e.g. all tiles and entities, including the player). An example of a completed `FancyGameView` is shown in Figure 2. The methods you must implement in this class are:

- `__init__(self, master: tk.Frame | tk.Tk, dimensions: tuple[int, int], size: tuple[int, int], **kwargs) -> None`: Sets up the `FancyGameView` to be an `AbstractGrid` with the appropriate dimensions and size, and creates an instance attribute of an empty dictionary to be used as an image cache.

- `display(self, maze: Grid, entities: Entities, player_position: Position ):` Clears the game view, then creates (on the `FancyGameView` instance itself) the images for the tiles and entities. If an entity is at a specific location, you may assume there is a `FLOOR` tile undeneath. If an entity is at a position, the tile image should be rendered beneath the entity image. **You must use the `get_image` function from `a3_support.py` to create your images.**

Figure 2: Example of a FancyGameView partway through a game.

6.2 FancyStatsView

`FancyStatsView` should inherit from `AbstractGrid` (see `a3_support.py`). It is a grid with 3 rows and 3 columns. The top row displays the text 'Player Stats' in a bold font in the second column. The second row displays titles for the stats, and the third row displays the values for those stats. The `FancyStatsView` should span the entire width of the game and shop combined. An example of a completed`FancyStatsView` in the game is shown in Figure 3. The methods you must implement in this class are:

- `__init__(self, master: tk.Tk | tk.Frame) -> None`: Sets up this `FancyStatsView` to be an `AbstractGrid` with the appropriate number of rows and columns, and the appropriate width and height (see `a3_support.py`).

- `draw_stats(self, moves_remaining: int, strength: int, money: int) -> None`: Clears the `FancyStatsView` and redraws it to display the provided moves remaining, strength, and money. E.g. in Figure 3, this method was called with `moves_remaining` = 10, `strength` = 4, and `money` = 7.

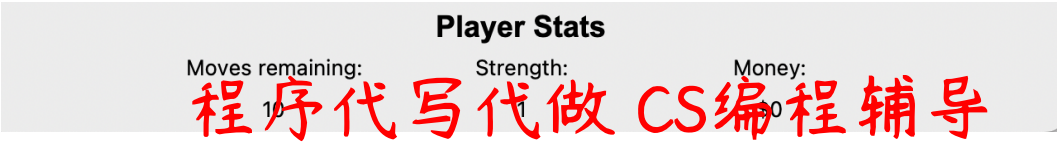

Figure 3: `FancyStatsView` after redrawing with moves_remaining set to 10, strength set to 4, and money set to 7.

6.3 Shop

`Shop` should inherit fro `Shop` is a frame displaying relevant information and buttons for all the buya e (see the `get_shop_items` method in `SokobanModel`). The `Shop` should contai and a frame for each buyable item (each potion). Each item's frame should contain the following widgets, packed left to right:

- A label containing the name of the item and the cost to buy that item.
- A button for buying the item at the listed price. The callback for these buttons must be created in the controller (see `ExtraFancySokoban`) and passed to the `Shop` when calling `create_buyable_item` (see below).

See Figure 4 for an example of the shop interface.

The methods that you must implement in this class are:

- `__init__(self, master: tk.Frame) -> None`: Sets up the shop to act like a `tk.Frame` and to have a title label at the top in bold font. Note that you are not required to create the item frames and internal widgets here.
- `create_buyable_item( self, item: str, amount: int, callback: Callable[[], None] ) -> None`: Create a new item in this shop. That is, this method creates a new frame within the shop frame and then creates a label and button within that child frame. The button should be bound to the provided `callback`.

Note: Handling callbacks is an advanced task. These callbacks will be created within the controller class, as this is the only place where you have access to the required modelling information. Start this task by trying to render display correctly, without the callbacks. Then integrate these views into the game before working on the callbacks.

6.4 FancySokobanView

The `FancySokobanView` class provides a wrapper around the smaller GUI components you have just built, and provides methods through which the controller can update these components.

The methods that you must implement in this class are:

- `__init__(self, master: tk.Tk, dimensions: tuple[int, int], size: tuple[int, int]) -> None`: Sets up a new `FancySokobanView` instance. This includes creating the title banner, setting the title on the window, and instantiating and packing the three widgets described earlier in this task sheet.
- `display_game( self, maze: Grid, entities: Entities, player_position: Position ) -> None`: Clears and redraws the game view.
- `display_stats(self, moves: int, strength: int, money: int) -> None`: Clears and redraws the stats view.

Figure 4: Shop interface.

- `create_shop_items( self, shop_items: dict[str, int], button_callback: Callable[[str], None] | None = None ) -> None`: Creates all the buyable items in the shop. `shop_items` maps item ids (result of calling `get_type` on the item entity) to price. For each of these items, the callback given to `create_buyable_item` in `Shop` should be a function which requires no *positional* arguments and calls `button_callback` with the item id as an argument. Note: if you create your callback within a loop using a lambda function, you may need to include a *keyword* argument with a default value of the specific item's id in order to prevent Python from using the last item for all buttons.

6.5 ExtraFancySokoban

`ExtraFancySokoban` is the controller class for the overall game. The controller is responsible for creating and maintaining instances of the model and view classes, event handling, and facilitating communication between the model and view classes. Figure 1 provides an example of how the `ExtraFancySokoban` game should look. Certain events should cause behaviour as per Assignment 2 (note that this includes the ability to undo a move). You should not reimplement this behaviour, but rather use `attempt_move` method for the model. The methods that you must implement in this class are:

- `__init__(self, root: tk.Tk, maze_file: str) -> None`: Sets up the `ExtraFancySokoban` instance. This includes creating instances of `SokobanModel` and `SokobanView`, creating the shop items, binding keypress events to the relevant handler, and then redrawing the display to show the initial game state. When creating the shop items, you will need to create a function to pass to the `create_shop_items` method. This method should:

  1. Take an item id as a parameter
  2. Tells the model to attempt to buy that item
  3. Tells the entire view to redraw

n and loss messageboxes.

- `redraw(self) -> None`: Redraws the game view and stats view based on the current model state.
- `handle_keypress(self, event: tk.Event) -> None`: An event handler to be called when a keypress event occurs. Should tell the model to attempt the move as per the key pressed, and then redraw the view. If the game has been won or lost after the move, this method should cause a messagebox to display informing the user of the outcome and asking if they would like to play again (see Fig. 5). If the user selects yes, the game should be reset (i.e. reset the model and then redraw the view). If the user selects no, the program should terminate gracefully.

6.6 `play_game(root: tk.Tk, maze_file: str) -> None` function

The `play_game` function should be fairly short. You should:

1. Construct the controller instance using the given `maze_file` and the root `tk.Tk` parameter.
2. Ensure the root window stays opening listening for events (using `mainloop`).

Note that the tests will call this function to test your code, rather than main.

6.7 main function

The purpose of `main` is to allow you to test your own code. 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, and the path to any map file you like (e.g. `'maze_files/maze1.txt'`).

7 File Menu

The file menu should be called 'File' and contain two options; 'Save' and 'Load'. When a user selects the 'Save' option, they should be prompted with a file dialogue (you **must** use tkinter's `filedialog.asksaveasfilename` for this) to enter a name for the file. You must then save enough details of the game to this file in order to recreate the entire game state (except for money) if the player tries to load this file in. When a user selects the 'Load' option they should be prompted with a file dialogue (you **must** use tkinter's `filedialog.askopenfilename` for this) to select a file. You must then load in this game state as the new game state. After selecting a file, the view should immediately update to show the new game state.

7.1 Notes

1. You must save and load all game information **except for money**. You may assume that when loading a game, the player should have 0 money, even if they had money in the saved game.

2. If the player loa[illegible]fferent dimensions to the original game, the images in the game view[illegible]rder to still take up the full space allocated for the `FancyGameView`. [illegible]e this, you will likely need to add a method to your `FancyGameView` cl[illegible]eset the cache, and add a method to your `FancySokobanView` class to allow you[illegible]and set the new dimensions on the `FancyGameView` instance.

3. The format used for storing game details in saved files is up to you. Your save and load functionality must work together (i.e. you must be able to load files saved through the 'Save' option you provide) in order to achieve marks for this task.