CSSE1001 Assignment 3

Due 8pm, Friday 28th May 2021

1 Introduction

In Assignment 2 you implemented a text-based game, *EndOfDayz*. This game was intentionally structured in such a way that the modelling logic and the view classes for displaying content to the user could be modified fairly independently. In this assignment, you will exchange the text-based interface of *EndOfDayz* with a more sophisticated tkinter-based Graphical User Interface (GUI), and implement additional functionality. You can find information about tkinter in this introduction to tkinter[1] and in this tkinter reference[2].

In Assignment 3 the user is presented with a map, on which entities are represented by either coloured squares or images. As opposed to Assignment 2, where the step event was triggered by user input, in this assignment the step event is triggered every second for all entities *other* than the player. The player's behaviour is triggered by key presses.

1.1 Variations from Assignment 2

When the player picks up an item, that item is added to an inventory, but not immediately applied. The inventory is displayed to the right of the game map. The user can *activate* and *deactivate* items by left clicking on them in the inventory. When an item is *activated*, it should begin working, and its lifetime should decrease by 1 every second. An example of the final game is shown in Figure 1. Additonally, the player can no longer run into a zombie to remove it from the game.

2 Tips and Hints

This assignment is split into two tasks, with an additional task for CSSE7030 students. 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. If you are a CSSE7030 student, you should not attempt the CSSE7030 task until you have convinced yourself (through testing) that task 1 and task 2 both work relatively well.

[1] `https://web.archive.org/web/20171112065310/http://effbot.org/tkinterbook/`

[2] `https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html?fbclid=IwAR0MRN_QOr-A-dYrfoWW2NAFUlrjyoF02PRBUv63OCl3tVBgLvrTXR2NZJ8`

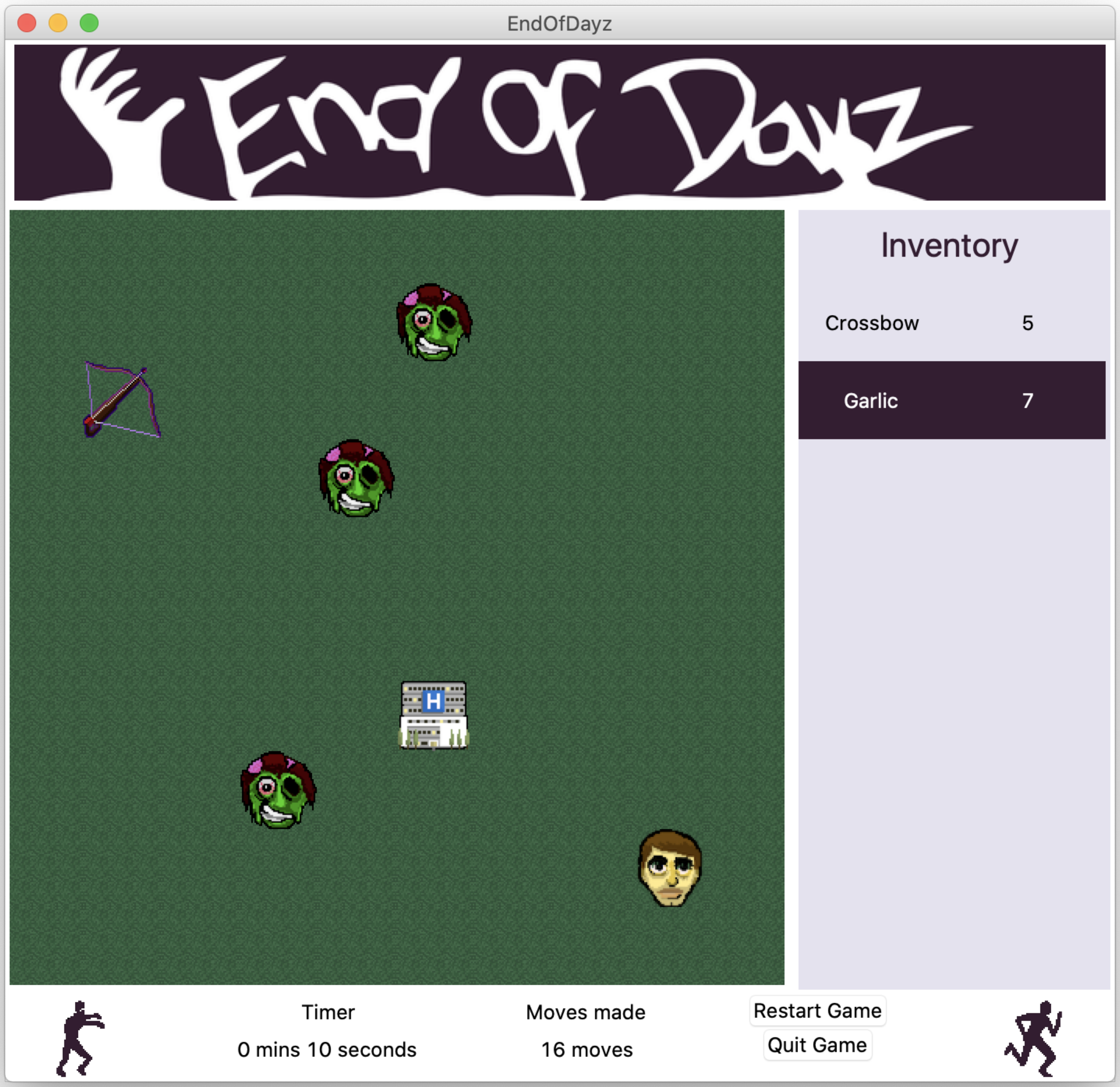


Figure 1: Example fully functional game (at the end of task 2).

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, comment it out before submitting your assignment. Your marker will **not** modify your code in order to test functionality.

You **must only** make use of libraries listed in Appendix A. Importing anything that is **not** in this list will result in a deduction of up to **100%**.

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

Task 1 requires you to implement a functional GUI-based version of *EndOfDayz*. At the end of this task your game should look like Figure 2. There are three major sections to the GUI; a heading label at the top, the game map (bottom left), and the inventory (bottom right).

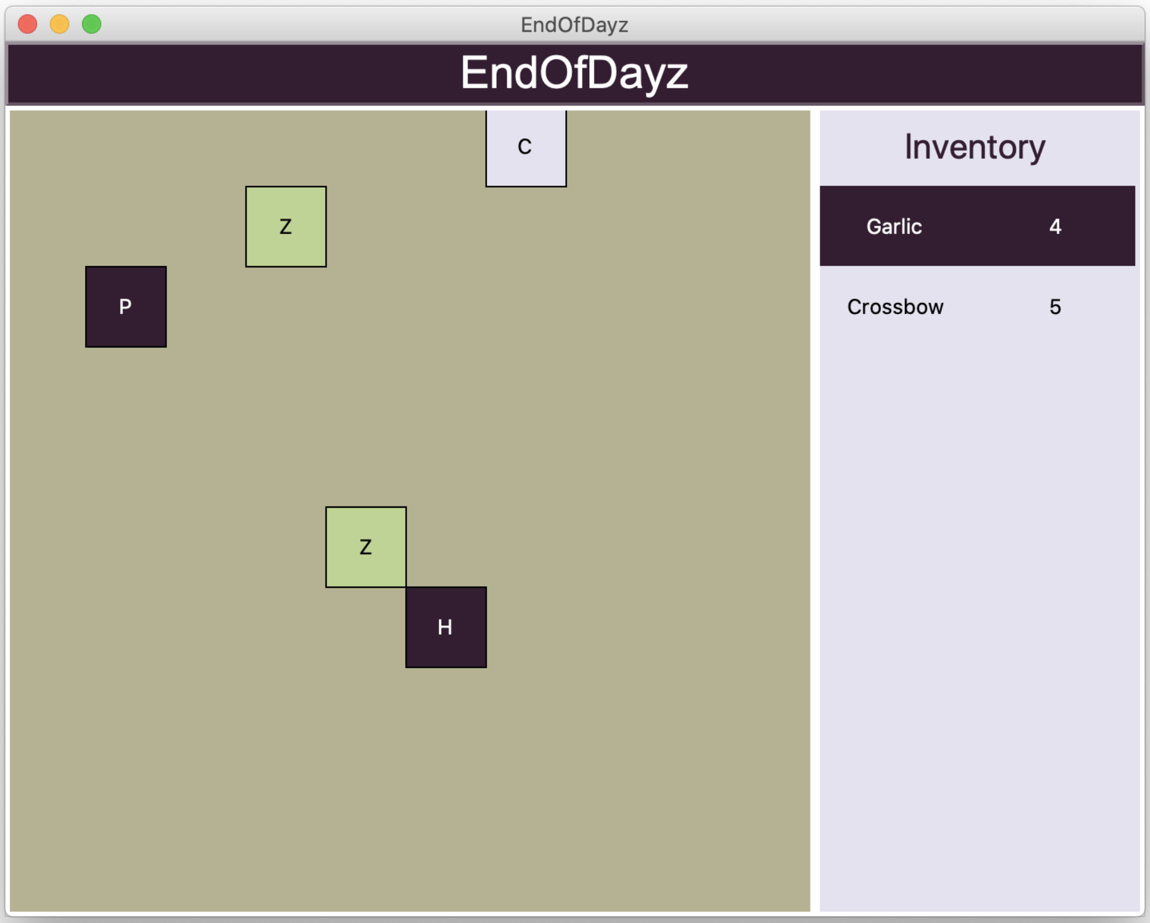


Figure 2: Example game at the end of task 1.

To complete this task you will need to implement various view classes, including a basic graphical interface, which handles most of the event handling that was previously handled in `TextInterface` and its subclasses. You **must** use the supplied Assignment 2 solution for most of the modelling, as this solution contains additional functionality that was not required in your Assignment 2. You may create additional modelling classes if they benefit your solution.

User input events should cause behaviour as per Table 1.

Event	**Behaviour**
Key Press: ‘w’	Player moves up one square if the map permits.
Key Press: ‘a’	Player moves left one square if the map permits.
Key Press: ‘s’	Player moves down one square if the map permits.
Key Press: ‘d’	Player moves right one square if the map permits.
Left click on inventory view	If the left click occurs anywhere on a row containing an item, the ‘activated’ status of that item is toggled (see Section 3.2.3 for details). If the left click occurs elsewhere nothing should happen (in particular no errors should occur).

Table 1: Task 1 events and their corresponding behaviours.

The player can move independently of the step events. That is, the player can move multiple times within the space of a single step event for the other entities.

When the player wins or loses the game they should be informed of the outcome via a messagebox, and asked whether they would like to play again. If the user opts to play again, the current game should be reset to its initial state and the user should be able to replay the game. If they opt not to play again the program should terminate. Hint: tkinter's `messagebox.askquestion` or `messagebox.askyesno` may be useful in achieving this.

The following sub-sections outline the recommended structure for your code. You will benefit from writing these classes in parallel, but you should still test individual methods as you write them. Within this section we outline some methods that may be useful to write in the view classes for task 1. Type hints are omitted, as it is up to you to determine what these should be. These lists are not necessarily complete and you may include additional methods if they improve the design of your classes. You may also need to add more methods to these classes for task 2 and/or the CSSE7030 task.

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 may modify the classes as you see fit, but any changes must be submitted with your assignment.

3.2 View Classes

You must implement view classes for the game map and the inventory. However, because these widgets can both be represented by grids of rectangles, you should create an abstract class to factor out the shared functionality.

3.2.1 AbstractGrid

`AbstractGrid` is an abstract view class which inherits from `tk.Canvas` and provides base functionality for other view classes. An `AbstractGrid` can be thought of as a grid with a set number of rows and columns, which supports creation of text at specific positions based on row and column. The number of rows may differ from the number of columns, and the cells may be non-square. You must define the constructor for the `AbstractGrid` class as:

- `__init__(self, master, rows, cols, width, height, **kwargs)`: The parameters `rows` and `cols` are the number of rows and columns in the grid, `width` and `height` are the width and height of the grid (in pixels) and `**kwargs` signifies that any additional named arguments supported by `tk.Canvas` should also be supported by `AbstractGrid`.

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

- `get_bbox(self, position)`: Returns the bounding box for the (row, column) position; this is a tuple containing information about the pixel positions of the edges of the shape, in the form `(x_min, y_min, x_max, y_max)`.
- `pixel_to_position(self, pixel)`: Converts the (x, y) pixel position (in graphics units) to a (row, column) position.

- `get_position_center(self, position)`: Gets the graphics coordinates for the center of the cell at the given (row, column) position.
- `annotate_position(self, position, text)`: Annotates the center of the cell at the given (row, column) position with the provided text.

3.2.2 BasicMap

`BasicMap` is a view class which inherits from `AbstractGrid`. Entities are drawn on the map using coloured rectangles at different (row, column) positions. You must annotate the rectangles of all entities with what they represent (as per Figure 2). You must use the `create_rectangle` and `create_text` methods from `tk.Canvas` to achieve this. The colours representing each entity are:

- Zombies: Light green (#B8D58E)
- Pickups: Light purple (#E5E1EF)
- Player and Hospital: Dark purple with white text (#371D33)
- Background: Light brown (#B5B28F)

Your program should work for reasonable map sizes, and you may assume that the map will always be square (i.e. the number of rows will always be equal to the number of columns). The `BasicMap` class should be instantiated as `BasicMap(master, size, **kwargs)`. The `size` parameter is the number of rows (= number of columns) in the grid. Each rectangle should be 50 pixels high and 50 pixels wide. You should set the background colour of the `BasicMap` instance by using the `kwargs`.

It may be useful to add the following method to the `BasicMap` class:

- `draw_entity(self, position, tile_type)`: Draws the entity with `tile_type` at the given position using a coloured rectangle with superimposed text identifying the entity.

3.2.3 InventoryView

`InventoryView` is a view class which inherits from `AbstractGrid` and displays the items the player has in their inventory. This class also provides a mechanism through which the user can activate an item held in the player's inventory. When a player picks up an item it is added to the inventory and displayed in the next free row in the inventory view, along with its maximum lifetime. Unlike in Assignment 2, the item is **not** immediately applied. Instead, when the user left clicks on the `InventoryView` in the row displaying that item, the item is:

- 'Activated' if the item was not already activated **and** no other item is currently active. Only one item may be active at any given time. Once activated, the row of the inventory view should be highlighted as per Figure 2, the lifetime should begin to decrease by 1 every game step (i.e. every second), and this change in lifetime should be reflected in the inventory view.
- 'Deactivated' if the item was activated. The item should no longer be highlighted, the lifetime countdown should stop, and the effects on the player should cease. A deactivated item can be reactivated, but the lifetime starts from where it left off.

Once the lifetime reaches 0, the item is no longer applied to the player and is removed from the `InventoryView`. Any items below the newly expired item should move up one row to avoid any empty space between items. The `InventoryView` should have two columns and a width set according to the `INVENTORY_WIDTH` constant in `constants.py`. The number of rows in the inventory (including the header row) should be equal to the number of rows in the game, and the height should be equal to the height of the game map. You may assume that the maximum number of items is always 1 less than the number of rows in the game.

In Assignment 2 the direction for firing the crossbow was determined by text input from the user. In Assignment 3, once the crossbow is activated the user should press the up, down, left, or right arrows on the keyboard to fire the crossbow in that direction. A single crossbow can be fired multiple times until its lifetime expires and it is removed from the inventory.

The constructor for the `InventoryView` class must be as follows:

- `__init__(self, master, rows, **kwargs)`: The parameter `rows` should be set to the number of rows in the game map.

The following are suggested methods to include in this class:

- `draw(self, inventory)`: Draws the inventory label and current items with their remaining lifetimes.
- `toggle_item_activation(self, pixel, inventory)`: Activates or deactivates the item (if one exists) in the row containing the pixel.

3.2.4 BasicGraphicalInterface

The `BasicGraphicalInterface` should manage the overall view (i.e. constructing the three major widgets) and event handling. The constructor should be defined as:

- `__init__(self, root, size)`: The parameter `root` represents the root window and `size` represents the number of rows (= number of columns) in the game map. This method should draw the title label, and instantiate and pack the `BasicMap` and `InventoryView`.

The following are suggested methods to include in this class:

- `_inventory_click(self, event, inventory)`: This method should be called when the user left clicks on inventory view. It must handle activating or deactivating the clicked item (if one exists) and update both the model and the view accordingly.
- `draw(self, game)`: Clears and redraws the view based on the current game state.
- `_move(self, game, direction)`: Handles moving the player and redrawing the game. It may be easiest to create a new method to handle the `'<KeyPress>'` event, which calls `_move` with the relevant arguments.
- `_step(self, game)`: The `_step` method is called every second. This method triggers the step method for the game and updates the view accordingly. Note: The `.after` method for tkinter widgets may be useful when trying to get this method to run every second.
- `play(self, game)`: Binds events and initialises gameplay. This method will need to be called on the instantiated `BasicGraphicalInterface` in `main` to commence gameplay.

3.3 main function

The `main` function is provided as part of the support code, it:

1. Loads in the game file specified by the `MAP_FILE` constant in `constants.py` as an `AdvancedGame`. Note that the displayed map must change appropriately to reflect the map in the file when this constant is changed.
2. Constructs the root `tk.Tk` instance.
3. Constructs the interface instance.
4. Causes gameplay to commence.

4 Task 2: Images, StatusBar, File Menu, and High Scores

Task 2 requires you to add additional features to enhance the game's look and functionality. Figure 1 gives an example of the game at the end of task 2.

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. In order to do this, it is *strongly* recommended that you create a new interface class that extends the functionality of the `BasicGraphicalInterface` class.

4.1 StatusBar and Banner

Add a `StatusBar` class that inherits from `tk.Frame`. In this frame, you should include:

- The chaser and chasee images (see images folder).
- A game timer displaying the number of minutes and seconds the user has been playing the current game.
- A moves counter, displaying how many moves the player has made in the current game.
- A 'Quit Game' button, which ends the program.
- A 'Restart 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 appeared at the start of the game. Clicking the 'Restart Game' button after game play is finished should start a new game.

You should add the banner image at the top of the interface, where in task 1 there was a title label. **For full marks, the layout of the status bar and positioning of the banner must be as per Figure 1.** You may find the Pillow library useful for resizing the banner image.

4.2 Images

Create a new view class, `ImageMap`, that *extends* your existing `BasicMap` class. This class should behave similarly to `BasicMap`, except that images should be used to display each square rather than rectangles (see the provided images folder). The game map should be set up as an `ImageMap` in task 2, but you should still provide a functional `BasicMap` class that allows us to test your task 1 functionality when `TASK=1`.

4.3 File Menu

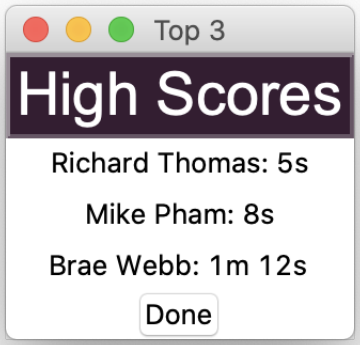
Add a file menu with the options described in Table 2. Note that on Windows this will appear at the top of the game window, whereas on Mac this will appear at the top of your screen. For saving and loading files, you must design an appropriate file format to store information about game state. You may use any format you like, as long as your save and load functionality work together to allow the current state of the game to be restored. 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**
Restart game	Restart the current game to its initial state.
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.
Quit	Prompt the player via a 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.

4.4 High Scores

To complete this task, you must add a 'High scores' option to the file menu. Selecting this option should create a top level window displaying an ordered leaderboard of the best (i.e. lowest) time achieved by users in the game (up to the top 3); see Figure 3. These scores should persist even if the app is run again.



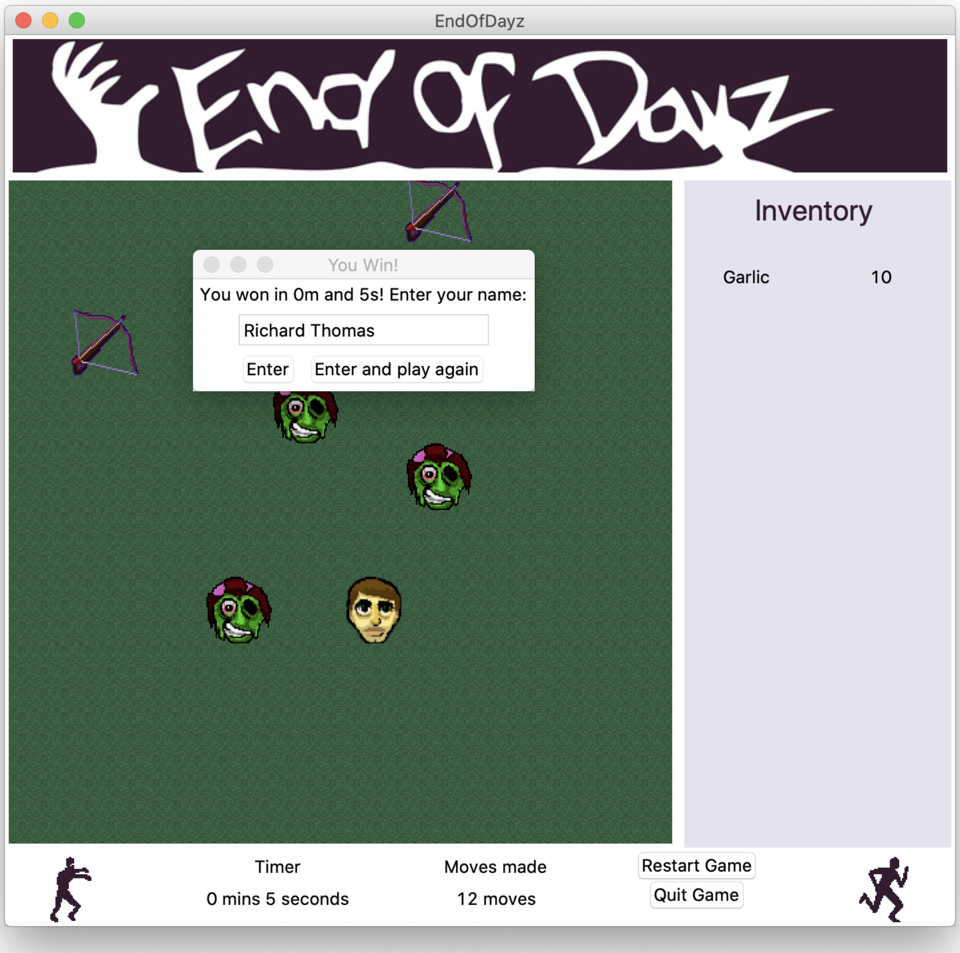
Richard Thomas: 5s

Mike Pham: 8s

Brae Webb: 1m 12s

Figure 3: High score window.

When the player wins or loses all entities should stop moving and the game timer should be stopped. You must inform the player of their outcome and, if the player has won, you must prompt them for their name to display next to their score (game time in seconds) in the high scores table if they are within the top 3; see Figure 4. Regardless of the outcome (win or loss) you must enable the user to play again (i.e. restart the game). If the user opts not to play again, the window should remain open, displaying the final game state. You will need to write high score information to a file (see `constants.py` for the high scores filename), 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.



You won in 0m and 5s! Enter your name:

Richard Thomas

Garlic	10

0 mins 5 seconds

12 moves

Figure 4: Prompt for name on game win.

5 CSSE7030 Task: Animations and Time Machines

All CSSE7030 tasks are required to only be visible when the `TASK` constant in `constants.py` is set to 3. Like with task 2, you should create a new interface class that extends the functionality of your task 2 interface class.

5.1 Animations

For this task you are required to animate the firing of the crossbow. When the crossbow is fired at a zombie the arrow image should be oriented from the player in the direction that it is being fired, and should move in that direction one square at a time (fairly quickly; the exact speed is up to you but it needs to be reasonable). The arrow should not fire if there is no zombie in its path at the time of firing. If the arrow reaches the zombie, the zombie should immediately be removed from both the model and view (i.e. you should not wait until the next game step to show this update on the view). If the zombie was in the path of the arrow but moves out of the way *before* the arrow reaches it, then it has avoided the arrow and should not be removed. You should enable multiple arrows to be fired in succession from the same crossbow (i.e. you should provide support for multiple animations to display at once).

5.2 Time Machines

You are required to add an extra pickup item, `TimeMachine`. This item has an infinite lifetime, but each instance can only be applied to the player once. If the player runs into a zombie while holding a time machine, the player will 'use' their time machine to travel back 5 steps (here, steps refers to player moves, rather than game timesteps). If there have been fewer than 5 steps in the current game, the game is essentially reset. Once the time machine has been used, the game (including the status bar) should be identical to how it was 5 steps in the past, and one time machine should be removed from the inventory. You will need to source or create an appropriate image for the time machines and submit this with your solution. The `display` method for the `TimeMachine` class must return the `TIME_MACHINE` constant from the `constants.py` file, as we will test your code by using the character in this constant to represent time machines in the game file.

6 Marks Breakdown

Your total mark will be made up of functionality and style marks. Functionality marks are worth 14 of the 20 available marks for CSSE1001 students and 20 of the 25 available marks for CSSE7030 students. Style marks are worth the remaining marks and will be assessed according to the criteria provided in Section 9. Your style mark will be calculated according to:

$$\text{CSSE1001 Final style mark} = \text{Style rubric mark} * \min(7, \text{Functionality mark})/7$$
$$\text{CSSE7030 Final style mark} = \text{Style rubric mark} * \min(15, \text{Functionality mark})/15$$

Your assignment will be marked by a tutor who will run your `a3.py` file and evaluate the completeness and correctness of the tasks you have implemented, and the quality of your code style.

Table 3 outlines the breakdown of functionality marks for the assignment.

Feature	**CSSE1001 Marks**	**CSSE7030 Marks**
Display Game GUI	2	2
Player Movement	2	2
Zombie Movement	2	2
Activate Inventory Item	1	1
Fire Crossbow	2	2
StatusBar and Banner	1.5	1.5
Images	1	1
File Menu	1.5	1.5
High Scores	1	1
Arrow Animation	0	4
Time Machine	0	2
Total	14	20

Table 3: Marks breakdown.

7 Assignment Submission

Your assignment must be submitted via the assignment three submission link on Gradescope. You are encouraged to separate your code into multiple files (in a logical way), but the main file that your tutor will run to execute your program **must be named** `a3.py`. You do not need to resubmit any files supplied to you (e.g. the images or map files). You do not need to resubmit `a2_solution.py`, **unless** you have made changes to the model. You should not have made significant changes to the `a2_solution.py` file (i.e. you should **not** be writing all of your code in this file).

Late submission of the assignment will **not** be accepted. In the event of exceptional circumstances, you may submit a request for an extension following the process described in the course profile. All requests for extension, including supporting documentation, must be submitted **at least 48 hours prior** to the submission deadline.

8 Appendices

8.1 Appendix A: Permitted Libraries

You will need the following libraries to implement a working solution:

1. tkinter
2. tkinter.messagebox
3. PIL
4. random (imported in the assignment 2 solution)

You may import the following libraries if you wish (but do not need them to create a working solution):

1. math
2. typing

Use of any other libraries (*including* built-in libraries) **is not permitted** and will be penalised by a deduction of up to **100%** of the Assignment 3 mark.

9 Style Rubric

	Proficient	**Competent**	**Novice**
Readability (20%)	All code is clear and easy to read. Identifier names are descriptive and informative. Whitespace and formatting enhance ease of comprehending logic. Code adheres to style guide.	The code is fairly easy to read. Most identifier names are descriptive and informative. Whitespace and formatting usually enhances ease of comprehending logic. Code has only a few, minor, breaches of the style guide.	Parts of the code are difficult to follow. Several identifier names are not descriptive or informative. In places, whitespace and formatting detract from comprehending logic. Parts of the code do not adhere to style guide.
Documentation (25%)	Almost all comments enhance comprehensibility of the code and design. All modules, classes, methods and functions are clearly and concisely described via informative and complete docstrings. All important or complex blocks of logic are clarifed by informative comments. No comments are extraneous.	Most comments enhance comprehensibility of the code and design. Most modules, classes, methods and functions are clearly and concisely described via informative and complete docstrings. Most important or complex blocks of logic are clarifed by informative comments. Almost no comments are extraneous.	Only some comments enhance comprehensibility of the code and design. Some modules, classes, methods and functions are clearly and concisely described via informative and complete docstrings. Some important or complex blocks of logic are clarifed by informative comments. Some comments are extraneous.
Algorithmic Logic & Design (25%)	Code is modular, minimising duplication of logic and keeping data as local as possible. Symbolic constants are used to minimise dependency on literal values. Logic is expressed succinctly and clearly through the use of appropriate control structures.	Most code is modular, minimising duplication of logic and keeping data as local as possible. Symbolic constants are usually used to minimise dependency on literal values. Logic is usually expressed clearly through the use of appropriate control structures.	Some code is not modular, resulting in duplication of logic or excessive sharing of data. Symbolic constants are only occassionally used to minimise dependency on literal values. Some logic is not clear or uses inappropriate control structures.
Object-Oriented Program Structure (30%)	Almost all classes represent single, self-contained, concepts in the program. The GUI's view and control logic is clearly separated from the model. Objects do not break encapsulation by accessing attributes of other objects. Public interfaces of classes provide a simple and clear abstraction of how to use objects of the class. Inheritance simplifies the design. Polymorphism allows the design to be easily extended.	Most classes represent single, self-contained, concepts in the program. The GUI's view and control logic is mostly separated from the model. Objects do not break class interfaces by accessing attributes of objects of other class types. Public interfaces of most classes provide a clear abstraction of how to use objects of the class. Inheritance usually simplifies the design. Polymorphism allows the design to be extended in some ways.	Only some classes represent single, self-contained, concepts in the program. In parts of the program, the GUI's view and control logic are not clearly separated from the model. Objects break class interfaces by accessing attributes of objects of other class types. Many methods are made public, when they should be private, or a few attributes are public. Some use of inheritance complicates the design. Polymorphism does not allow the design to be extended easily.