Python Object Oriented Programming Maze Game Group Report

ACIT 2515, Apr. 5, 2021, Set B

Sahil Singh A01200986

Brian Lam A01236157

Raihan Kheraj A01196507

Cyrus Chan A01242596

Individual roles: *Loose roles*

Brian: Game logic programming, Code documentation, Server-side / data persistence, Unit testing, UML

Cyrus: Game logic programming, General app programming, Code documentation, UI programming, Unit testing

Raihan: Project management, Code documentation Report write-up, UML, General app programming, Unit testing

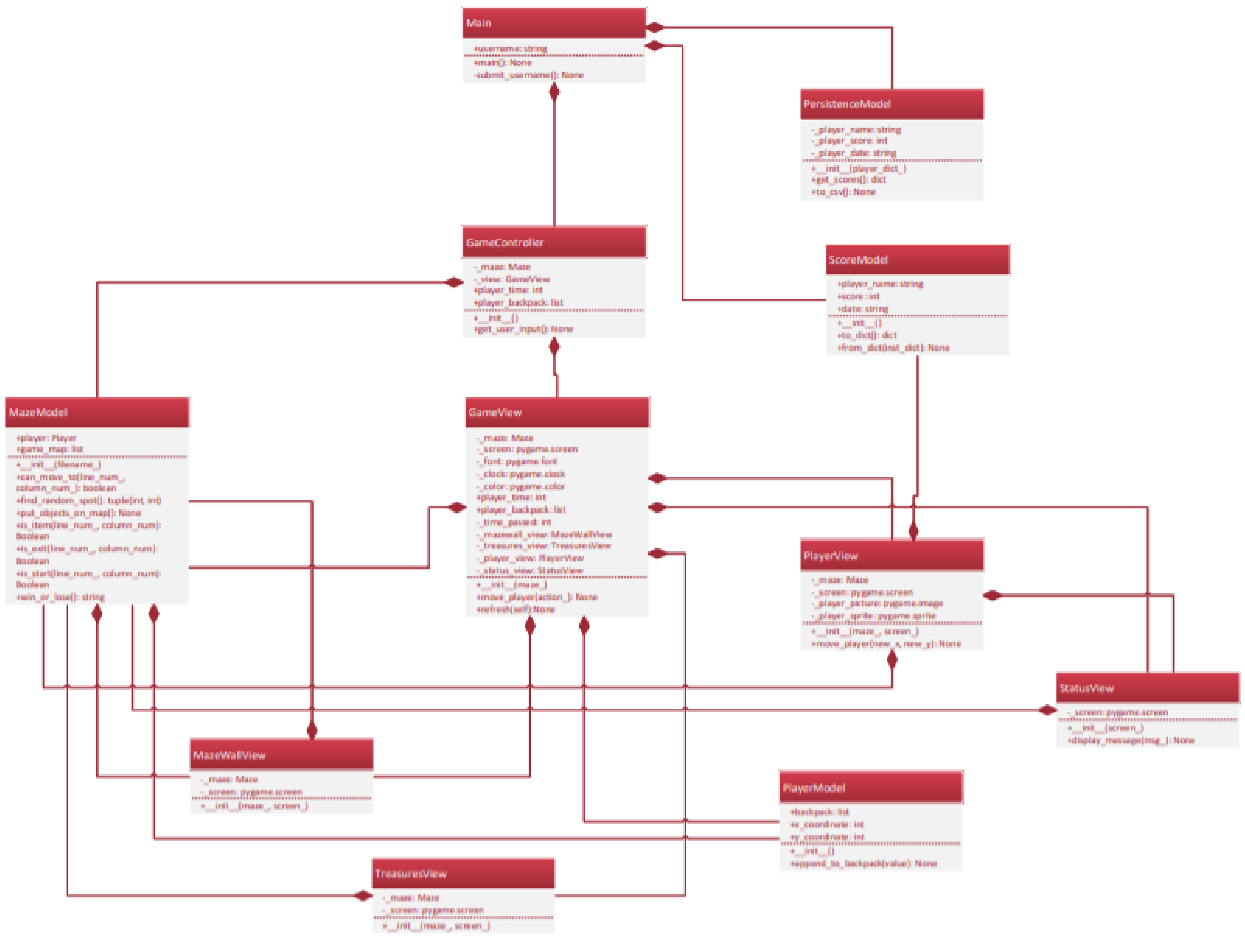
Sahil: UI asset preparation, General app programming, UI programming, Server-side / data persistence, Unit testing

Our group decided to approach the development of this project with equal responsibilities, and a vow to help each other in all aspects of the game. Each member had a hand in every feature, and was in charge of a functional area of the project.

Unit testing was implemented by all team members working together. Sahil spearheaded the data persistence, general app programming, and unit testing. He was responsible for choosing CSV as the persistence method, and taking charge of creating the associated functions to save data. Sahil and Brian chose CSV as the persistence method due to its simplicity and human readability.

Each member was responsible for proper code documentation; Raihan and Brian reviewed all comments to ensure consistency. Cyrus tackled unit testing, game logic, and UI programming. Cyrus did the research on tkinter and figured out how to pass the username through the tkinter GUI so it could be uploaded via the API. The UML was created by Raihan and Brian. As stated in the final project deliverable, we decided to document each test file instead of creating a test_plan.pdf.

UML (PDF is in documentation folder - less blurry)



Elaboration on reasoning behind code structure (1 paragraph – half a page)

We designed the maze game using an MVC pattern.

There is only one controller called game.py and it is responsible for collecting the keyboard input from the user and determining which direction to move the player.

The models include: maze, persistence, player, and score. Maze creates an instance of our maze and determines how the game operates. Persistence allows the user data to be saved if the game is won. Player stores information about the player in the game. Score stores information about: player name, score, and time of completion - Information that will be needed for the web API.

Finally, the views include: game, mazewall, player, status, treasures (a view for each item in the game). The Game view creates the other sub views in the game (each view is split into its own class). Game reacts to player movement and shows the player moving across the screen. Mazewall displays the maze based on an iterable list. Player displays the player object. Status shows whether or not the player has won the game. Treasures displays items placed randomly on the map.

MVC was chosen because it separates different functionalities of the program. All display related functions are placed in the views folder. Game Logic functions are placed in the models folder. Functions related to controlling the player are stored in the controllers folder.

For the web aspect of the project we have three folders, these include: static, templates, web_models. Static and template are made because of Flask's functionality, static contains a css file, while templates contain html files using jinja.

The web models folder contains two models: score manager, and score. Score contains the same data as its counterpart in maze, the same can be said about score manager. One key difference between the score managers is how they handle csv data. The score manager inside the web directory does not use Dictreader while the one inside maze does.

Player score calculation

The only way to generate a score in our game is to collect all 4 items within the specified 60 second time limit. If the user does not collect all 4 items, or runs out of time, they will be not given a score and therefore not added to the game's high score web page. The score is calculated by multiplying the time remaining by 25. Hence, the more time that is remaining, or the faster that the maze game is completed, the higher the score that the player will receive.

Persisting data with CSV - Web API

Our Web API stores data using CSV. In our web directory, our score_manager.py handles the persisting of data. We chose CSV because it is easily readable by humans and allows for a much easier time debugging. Due to the simple nature of the data, a more complex and organized structure method is unnecessary.

Bonus features implemented

Timer:

We implemented a countdown timer (60 seconds) that users have to race against to collect all the items in the maze. The timer is implemented in the game view, and refreshed with a function inside that file at 30 FPS. A surface is blit on the game to show the timer on the bottom to the user. If the timer reaches 0, the system will exit and display a status message to the user "You LOSE – Ran out of time". The status has its own view, and displays whichever message is passed to it.

Player username:

We decided to use the tkinter GUI popup before the user plays the game so they can submit a username. The given username is saved to a global variable, and then a player object can be created with that username and data from the game (score and date). Then that player object can be sent to the web API and displayed on the web page. If the user runs out of time or does not collect all of the items, they will not have their score saved. In other words, the user must win to receive a score, which is then saved in the CSV file, and later displayed on the game's high score web page.

CSS for web page and clear high score button:

We added some functionality and design to the web page using CSS. The webpage now has table formatting, custom colors, a button to clear all of the high scores, and a font that is representative of the game's theme. If there are no scores in the CSV file, the webpage will let the user know this. We could not figure out how to refresh the webpage with flask on the press of the button, so the user will have to refresh the page after clicking the button to remove the high scores.

View for every item:

Each item has their own view that is controlled by one super view (GameView).