CPEN 333: Final Project Report

Group G6 : Muntakim Rahman and Tomaz Zlindra

December 6th, 2024

1 Part I : Multithreaded Game

1.1 Requirements and Constraints

We were provided the template with skeleton code (i.e. classes, methods with docstrings) for the implementation of the Snake Game. The key design structure entailed the following requirement:

Use Python's *queue* module to ensure multi-producer, multi-consumer achieves synchronization between threads (i.e. add *dict* items to the queue for "game_over", "score", "prey", "move").

To program a responsive and thread-safe Snake Game, we use the template data fields and methods for the shown interactions.

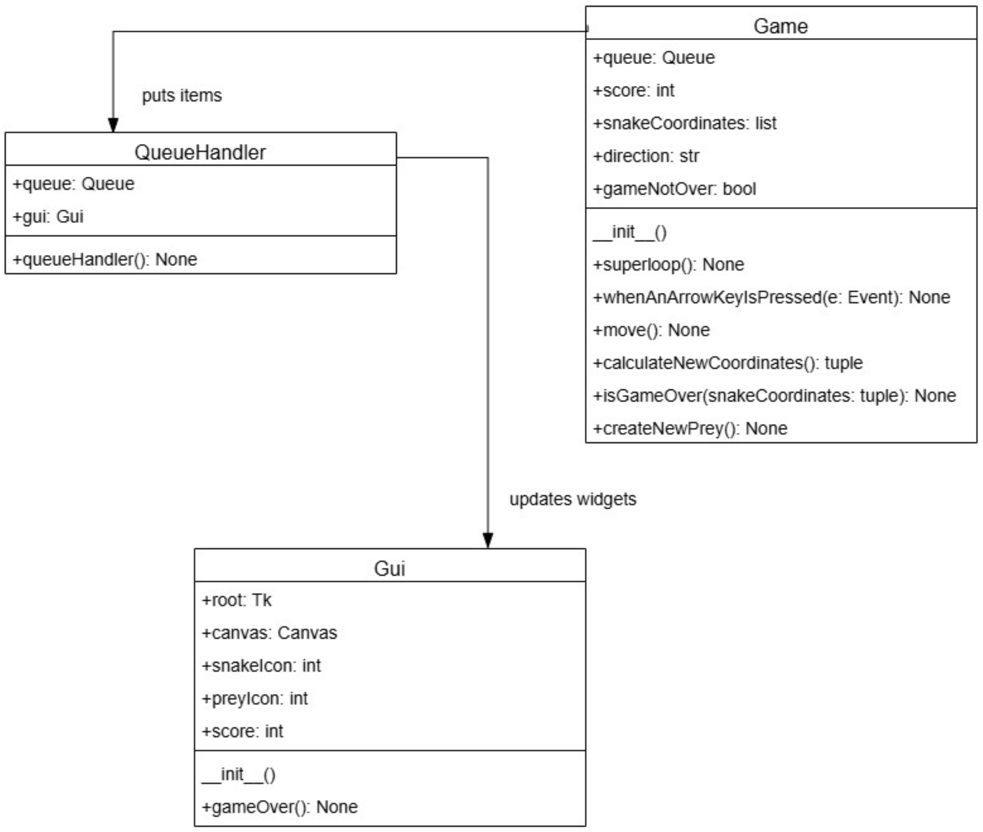

Figure 1: Part 1 : Interactions Between QueueHandler, Game, Gui

1.2 Structure

The general structure for Part I was already provided.
We only needed to complete the methods for the "Game" class, such as the superloop, move, calculateNewCoordinates, isGameOver and is createNewPrey methods.

1.3 Implementation

1.3.1 New Snake Coordinates

This method is very straightforward. It checks the direction by accessing `self.direction` to determine which direction the head of the snake should travel, and adding/subtracting the `SNAKE_ICON_WIDTH` by the current x or y coordinate.

1.3.2 Prey Generation

To generate the prey, we used the `random` library to generate a random integer for the x direction and the y direction on the Tkinter canvas. We added a small margin so the prey did not generate on or outside the edges of the window.

1.3.3 Prey Capture

One of the primary challenges we faced was implementing the prey capture logic. Since prey can spawn at any location on the Tkinter canvas, iterating through a list of tuples to compare coordinates is both resource-intensive and time-consuming. To optimize this process and eliminate the need for nested "for" loops, we calculated the corner coordinates of both the prey and the snake's head. We then checked whether any point of the snake's head fell within the prey's boundaries, as illustrated in figure 2. This method works effectively when the prey is larger than the snake. In cases where the snake is significantly larger than the prey, we reversed the logic: if any point of the prey fell within the snake's boundaries, it would be counted as a collision.

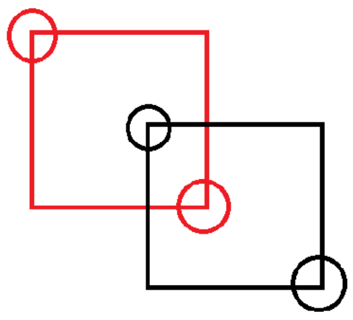

Figure 2: Prey Capture Criteria : Edge Coordinate Of Prey/Snake Must Be Contained

This solution mitigates the amount of recursion performed, while still detecting snake and prey collision.

1.3.4 Move

This method completes most the important game logic once the snake moves. Firstly, we had to check whether the prey is captured. If true, we added one to the score, increased the length of the snake, and created a new prey. Otherwise, the length of the snake would not change. Checking if the game is over is also necessary. Tasks are added to the queue are added here so the QueueHandler could manage edits to the GUI.

1.3.5 Game Over Logic

The game ends once:

- the snake's head either goes out of bounds, OR
- the snake runs into itself.

We checked whether the coordinates of the snake's head are outside the bounds of the Tkinter canvas or if the head coordinate is the same as any of the other snake coordinates.

2 Part I Alternative

2.1 Modified Approach

For the Part I Alternative, our one main modification to Part I was to remove the use of the QueueHandler class. This was replaced by the GUI class scheduling GUI updates every 100ms using `self.root.after(100, self.update)`. Since the GUI and move thread are performing operations simultaneously, we had to add locks to ensure data was not being modified as the changes were being updated by the GUI. The GUI processes used non-blocking locks so none of the GUI methods stalled while waiting for the move thread to either read or write the data.