COMPENG 2SH4 Project – Peer Evaluation [30 Marks]

Your Team Members Kishoban Ravendran (ravendk), Ratish Gupta (guptar76)

Team Members Evaluated Pusapatk, alir49

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions. Completing the peer code evaluation on time will earn your team a total of **30 marks**. Do not exceed 2 paragraphs per question.

Peer Code Review: OOD Quality

1. **[3 marks]** Examine the main logic in the main program loop. Can you easily interpret how the objects interact with each other in the program logic through the code? Comment on what you have observed, both positive and negative features.

   I can easily interpret how the objects interact with each other in the program logic, as each distinct feature is separated into distinct classes. There are pointers to the player, game mechs, and food classes, respectively, making for more modular code. In the run logic function, the appropriate member functions are accessed to update the player direction (using the player pointer), using the same logic found in PPA2 and PPA3. In the Draw Screen function, each of the pointers for the different classes work in tandem to draw contents onto the screen while adhering to the borders and other conditions. For example, random food generation is appropriately executed by only calling on the food pointer to carry out food generation if it's within the game board's borders, while utilizing Boolean conditions to let the program know when to begin printing the snake body using the player pointer. In this case, the objects interact with each to ensure that there is no overlap or missing contents when the game board is first generated.

2. **[3 marks]** Quickly summarize in point form the pros and cons of the C++ OOD approach in the project versus the C procedural design approach in PPA3.

Pros:

- **Modularity**: Code is structured into distinct classes, making it easier to manage and understand.
- **Reusability**: Classes and objects can be reused in other projects or extended for new features without major changes.
- **Maintainability**: Organized design allows for easier debugging, updates, and scaling.
- **Encapsulation**: Data and functions are bundled within classes, ensuring better security and control.

Cons:

- **Complexity**: More components (e.g., classes, objects) increase the code's complexity.
- **Memory Management**: Requires careful handling of dynamic memory allocation and deallocation (e.g., destructors).
- **Verbose**: Writing class definitions and handling object interactions adds extra overhead.

Peer Code Review: Code Quality

1. **[3 marks]** Does the code offer sufficient comments, or deploys sufficient self-documenting coding style, to help you understand the code functionality more efficiently? If any shortcoming is observed, discuss how you would improve it.
   This code offers comments that act as anecdotes within each function that describe what each of its mechanics are in a way that helps anyone reading through the code properly understand it. There are only comments provided if necessary (with a few exceptions). For example, in the generateFood function, there are an extensive number of comments detailing mechanics such as the use of rand() to generate the food in a random position within the x-border and y-border along with the use of a Boolean condition to validate that it has been generated in an appropriate position.
2. **[3 marks]** Does the code follow good indentation, add sensible white spaces, and deploys newline formatting for better readability? If any shortcoming is observed, discuss how you would improve it.
   Through looking at the entire code, it is seen that there is consistently good indentation, appropriate white spaces and it properly utilizes newline formatting.

Peer Code Review: Quick Functional Evaluation

1. **[3 marks]** Does the Snake Game offer smooth, bug-free playing experience? Document any buggy features and use your COMPENG 2SH4 programming knowledge to propose the possible root cause and the potential debugging approaches you'd recommend the other team to deploy. (NOT a debugging report, just technical user feedback)
   The Snake Game does offer a smooth experience and no bugs were seen after playing through the game a few times.
2. **[3 marks]** Does the Snake Game cause memory leak? If yes, provide a digest of the memory profiling report and identify the possible root cause(s) of the memory leakage.
   Dr. Memory did not run and was giving errors. The project however seems to have all memory allocation and deallocation functions properly called. Hence, it can be assumed that there is 0 bytes of memory leaks.

Project Reflection

Recall the unusual objPos class design with the additional Pos struct. After reviewing the other team's implementation in addition to yours, reflect on the following questions:

1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?
   Yes, I feel that the objPos class is sensible, since it holds all necessary variables for each object, such as the coordinate and the symbol.
2. **[4 marks]** If yes, discuss about an alternative objPos class design that you believe is relatively counterintuitive than the one in this project. If not, explain how you'd improve the object design. <u>You are expected to facilitate the discussion with UML diagram.</u>

   One alternative design for the objPos class could be to separate the coordinate i.e. positions and symbol into two different classes, creating an approach where each class represents a single responsibility. This approach will help in handling the symbols and position of the object correctly, especially when handling more complex printable characters, such as the UTF-8 schema.

```
+ ########################################+
|                 ObjPos                 |
+#########################################+
| - pos: Position                          |
| - symbol: Symbol                         |
+--------------------------------+
| + objPos()                               |
| + objPos(Position, Symbol)               |
| + objPos(const objPos&)                  |
| + operator=(const objPos&): objPos&      |
| + ~objPos()                              |
| + setsymbol(Symbol)                      |
| + setObjPos(objPos)                      |
| + setObjPos(Position, Symbol)            |
| + getObjPos(): objPos                    |
| + getSymbol(): char                      |
| + getSymbolIfPosEqual(const objPos*): char |
| + isPosEqual(const objPos*): bool        |
+--------------------------------+
```

Here, the symbol and and position classes are implemented.