COMPENG 2SH4 Project – Peer Evaluation [30 Marks]

Your Team Members Yahia Hassanen Andy Tang

Team Members Evaluated Edwin He Aaryaman Kumar

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.

The main program loop is logically structured and easy to interpret, with clear sequencing and separation of responsibilities. The functions have intuitive names to what they do, examples include **`hasWonGame`**, **`hasLostGame`**. Not only are the functions names clear but they are also not repetitive or too long. However, there could have been more effort into adding more comments through this specific block of code, it would've allowed for even easier understanding for anyone who looks at the code, although it is good as is. Overall there isn't much else to work on as the main function is very short. Bonus for remembering to return 0!

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 of C++ OOD include encapsulation, scalability, and reusability, while cons include general added complexity. The C++ OOD approach allows for the game logic to be organized into classes like **`Player`**, **`GameMechs`**, and **`objPosArrayList`**, which encapsulate related data and behavior, improving modularity and readability. As a result, it is easier to add extra features and upgrade various aspects of the game, such as the bonus in this case, of having special food and more than one food. This ties back to reusability as the methods we create can be used in other contexts or projects, not just this game. However, this comes at the expense of overall code complexity, as this approach was much more complicated and prone to errors and bugs when compared to the C procedural approach adopted in PPA3, where all the logic was in one shared file, allowing the debugging process to be much easier.

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.

   Yes, the code is well-commented throughout the program. Each block of code has its purpose written out, and the order of placement is logical and easy to follow. For example, movePlayer(), the comments of move body, remove head and add tail, update head based on position, and insert new head all follow a logical order.

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.

   Yes, the code does have good indentation and stylistic practices. This good indentation practices carries over to the comments as well, where they are lined up. It uses new lines to make the code very readable. One small addition would be to have an end game message, to make it absolutely clear that the game has ended either by death or by manual exit.

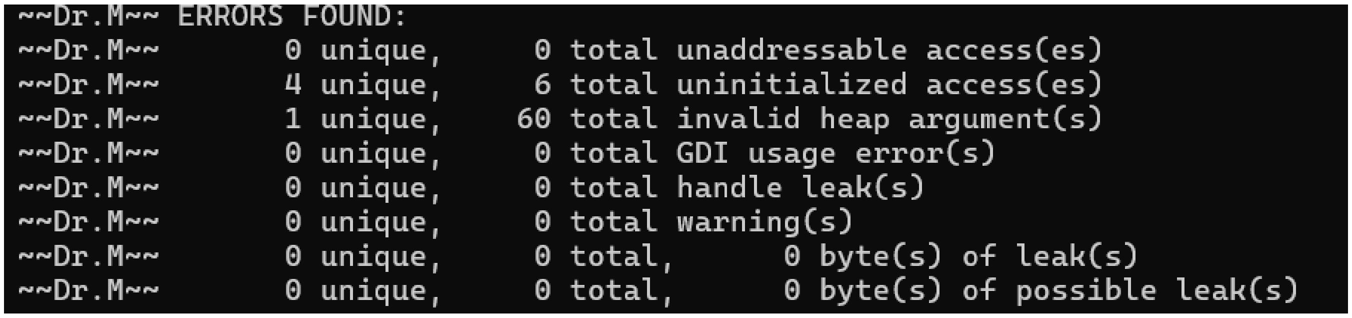
Peer Code Review: Quick Functional Evaluation

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)

   Overall, this program offers a relatively bug free experience. However, given that the body of the snake and the food are of the same character, it makes it slightly harder to differentiate between the two when the length of the snake is too long. To add, when the length grows to a very large amount, it may suddenly exit. This can be the result of a memory issue, or a buffer/out of bounds access issue.

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.
3.

   No according to the memory report there are no memory leaks!



```
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       4 unique,     6 total uninitialized access(es)
~~Dr.M~~       1 unique,    60 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total GDI usage error(s)
~~Dr.M~~       0 unique,     0 total handle leak(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
```

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?

   Similar to question 1.2. The compound object design of the **objPos** class, which includes the additional **Pos** struct to store coordinates, offers benefits such as encapsulation and potential reusability but comes with the drawback of added complexity. This design encapsulates the x and y coordinates within a dedicated struct, grouping related data logically and making it easier to extend functionality. It also promotes reusability, as the **Pos** struct could be used elsewhere in the project, making it versatile beyond just the **objPos** class. This implementation though is overall not sensible in our opinion as it introduces unnecessary complexity for a lightweight application like the Snake game. Managing Pos as a heap-allocated pointer increases the risk of memory leaks and makes debugging much much harder, as compared to storing x and y directly in **objPos**. In this context, a simpler design where x and y are stored directly within **objPos** would have been more efficient and easier to maintain.

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>

Given that the compound object design of **objPos** is not ideal for this project, an alternative, simplified design is one that directly stores x, y, and symbol in the **objPos** class itself, almost similar to PPA3 This approach reduces complexity, minimizes the risk of memory management errors, and improves performance by eliminating the indirection introduced by the heap-allocated Pos struct.

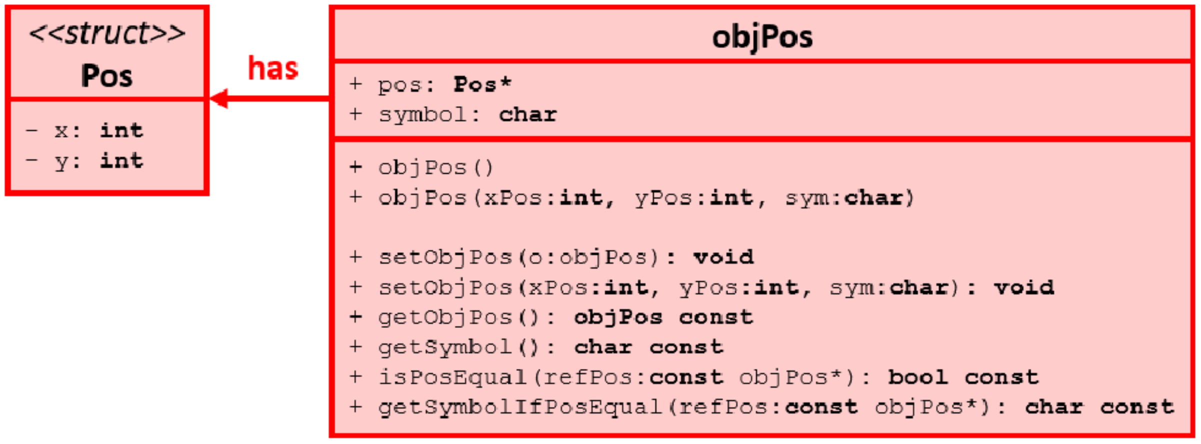

<<struct>>
Pos
- x: int
- y: int
has
objPos
+ pos: Pos*
+ symbol: char
+ objPos()
+ objPos(xPos:int, yPos:int, sym:char)
+ setObjPos(o:objPos): void
+ setObjPos(xPos:int, yPos:int, sym:char): void
+ getObjPos(): objPos const
+ getSymbol(): char const
+ isPosEqual(refPos:const objPos*): bool const
+ getSymbolIfPosEqual(refPos:const objPos*): char const

Figure 1. We see that x and y are stored separately from objPos. Our change would be to directly put it with symbol in the objPos class, to improve the performance of the program