COMPENG 2SH4 Project – Peer Evaluation [30 Marks]

Your Team Members	duam4, mustam26
	Git Team name - Hackstreet Boys
Team Members Evaluated	hasanz15, sarkeb1
	Git Team name – 404 Partner Not Found

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.
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.

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.
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.

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)
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.

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?
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>

Peer Code Review: OOD Quality

Within this team's code, it is easy to see how each object interacts with each other. In the main project file, when each method is called, its arguments are usually passed by value. This makes it easy to see the dependencies of each class within the main project file. However, OOD was not fully followed in this project. The logic to see if the player has collected a food object is within the main project file, but it should be implemented as a method in the player class. In addition, a global exit flag variable is used to decide when to end the game on top of the one stored in the GameMechs class. Only the GameMechs flag should be kept. This project uses OOD, however it is not realized to its full potential.

The C++ OOD approach allows for adaptability. Since all the logic is no longer in the main function alone and is spread across individual objects, it was a lot easier to adapt the code to complete other objectives in the future. For example, creating the food class in this project allowed for us to encapsulate all the logic required for food and its respective functions into one file, making it easier to manage and use. Some cons of the OOD approach are that it gets very complicated when an object contains other objects, and it becomes hard to trace what each of the objects are doing and their dependencies. Additionally, OOD depends on the assumption that the objects a class depends on are functioning perfectly and have no bugs. If this is not true, debugging becomes very difficult as there are many points of failure. Testing as you go is imperative for OOD.

Peer Code Review: Code Quality

With the group's code, the code is indented well and had plenty of white spaces and new line characters to ensure code readability. However, they provided excessive and unnecessary comments in obvious places and left out comments in important portions of the code. For example, in the food class, there are lines calling multiple functions that lack comments on what they do. An example being the lines that randomly create the coordinates for new food. Additionally, this team did not remove commented out code lines form previous iterations, which can make it confusing to understand which logic is being used and which is not. This group also changed the default end game key from the space bar to '!' without printing a UI message indicating this. This meant we did not know how to end the program, and we had to go into the code and find out the key that ends the game. A UI message indicating this change to the user is imperative. An additional minor improvement to make use of the predefined getter function .getChar() for an objPos instead of directly accessing the symbol data member. Although the symbol variable is defined publicly, it is better practice to use the getter function where needed. Making use of effective commenting and UI messages would greatly help both developers and users of this project.

Peer Code Review: Quick Functional Evaluation

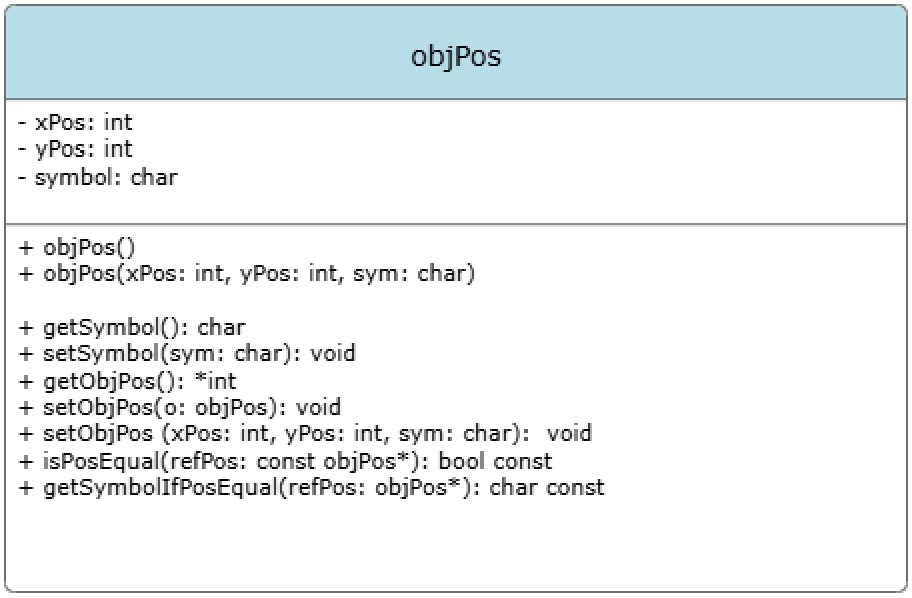
The game provides a nice smooth experience. There are no major bugs in the game, and it plays properly. Changing speed, the score increase, movement, and forcing the game end all work as expected. One minor bug is that if the snake collides with itself, during the end screen, the player crosses over themselves an additional space when they lose. This means that the FSM is allowed to run one more time after the game has ended before it terminates. This can be fixed by moving the logic to move the player before the logic for updating the player direction (and consequently checking for self collisions). Other than this small issue, the game runs perfectly.

The code did not have any leaks, and all heap data members were deleted correctly.

Project Reflection

We believe that the compound object design of objPos is not that sensible. It makes little sense to use a Pos struct to hold the X and Y coordinates of an object instead of just having the X and Y coordinates be private members of the objPos class. These data members are also all public, meaning they can be changed by any other part of the code at any time. Additionally, the getObjPos getter returns another objPos, meaning it is essentially useless in accessing the data members. Following proper OOD, the objPos class should have private data members that are interfaced with a getter (that allowed for the data members to be read properly) and a setter. Having objPos hold a pointer to a Pos struct is counterintuitive and makes for code that is harder to read.

As mentioned earlier, this design can be improved by having setters, as those can have logic to make sure the coordinates are set to something that is withing the gameboard, and getters, as those guarantee we don't accidentally modify the values store in objPos when we want to just access them. The getter would return a pointer to an int array that would hold the xPos, yPos, and the symbol (as an int). Below is the UML of the improved objPos class we've designed. The main changes are making the data members private and not in a struct, changing what the getObjPos() method returns, and having a set symbol setter.


objPos
- xPos: int
- yPos: int
- symbol: char
+ objPos()
+ objPos(xPos: int, yPos: int, sym: char)
+ getSymbol(): char
+ setSymbol(sym: char): void
+ getObjPos(): *int
+ setObjPos(o: objPos): void
+ setObjPos (xPos: int, yPos: int, sym: char): void
+ isPosEqual(refPos: const objPos*): bool const
+ getSymbolIfPosEqual(refPos: objPos*): char const