COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members _____chanc167_____ _____ fongt5_____

Team Members Evaluated _____harvev1_____ _____stepakJ_____

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 **25 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.

   Their code is easily interpreted due to appropriate use of OOD leading to a high level of organization. They effectively separated the respective methods into appropriate classes using easily understood names. The code follows appropriate OOD principles allowing the project.cpp file to contain very minimal lines as the programmers simply call the methods from their respective classes.

   The group implemented the collision logic within the move player method, which helps to promote conciseness of code in the main project file. However, keeping the collision logic within the movePlayer() method is unintuitive, and makes it so any future additional features to the collision logic may impair the clarity of the movePlayer() method. Moreover, a high-level overview of the main program loop is unable to identify where exactly collision is being handled, without having to dive deeper into different sections of the code. Therefore, it would be more sensible to implement the collision logic outside of the movePlayer() function.

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
   - OOD allows for better code modularization
     - Sections of code are broken down into smaller blocks, such as individual methods in individual classes, that can be tackled one at a time
     - It is therefore easier to divide tasks among multiple developers in larger scale projects
     - Code modularization saves development time down the line
   - OOD allows for greater abstraction of interactions in the program. For instance, once a class had been implemented, it is easy to use methods within the class without having to concern oneself with how the method works specifically
   - OOD allows the project to scale more efficiently. For instance, if more special fruits were to be added to the snake game, it would become beneficial to make use of principles of inheritance to create different kinds of fruits, that each inherit from a parent fruit class.

- OOD improves the readability of code, assuming the programmer aptly names their methods and classes. If you see, for example, the line of code if(player -> checkSelfCollision()), it is easy to understand that the if statement checks for the snake's collision to self, without the need to use a single comment.
    - Comments can thus be reserved for explaining implementation of more complex systems or methods. We can thus avoid over-commenting, which hinders code readability

Cons

- OOD typically requires more planning when used to realize a working program, as the programmer needs to have an understanding of how different objects interact with one another.
    - This comes with an initially longer development time
- As a project grows, it becomes increasingly difficult to make changes in an object-oriented design, especially when one must consider the many interdependencies between many classes
    - Debugging can become difficult, as changes in one class can have downstream effects that cause unwanted behaviour in seemingly separate classes
- If managed poorly, OOD can easily create unnecessary levels of complexity for the given task. For instance, we initially planned to create several children of a parent Food class, that could inherit its collision behaviour and have different effects on the score or the player. We quickly deemed it wasn't worth the effort for our use case, where we would only have one or two different types of Food.

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.

   In general, the code offers sufficient comments to understand code functionality. However, there are many instances where comments are unnecessarily included, which takes value and emphasis away from sections of code whose comments explain important functionality. For example, in the Player.cpp file, comments are included for getPlayerPos() to indicate it is a "Getter" that "Returns the reference to the playerPos array list." Comments are also used beside each case in the switch statement to reiterate what key is being handled. As a reader who is interested in understanding the code, these lines of code are rather self-explanatory and do not need commenting.

   Removing unnecessary comments will help a reader understand that, when comments are present in the code, attention should be diverted to that section of code, as there is likely important functionality that can be understood by reading the comment.

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.

   The code follows good indentation with a sensible use of white spaces and newlines. There is inconsistency in brace style used, which is understandable due to differing habits between team members.

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 runs very smooth and there are very few stutters when the screen is clear or the object changes. The wrap around logic and snake growth function as intended as well as the random food generation.

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.

   The group was successful in dynamically managing their memory. No memory leakage was observed.

Project Reflection

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

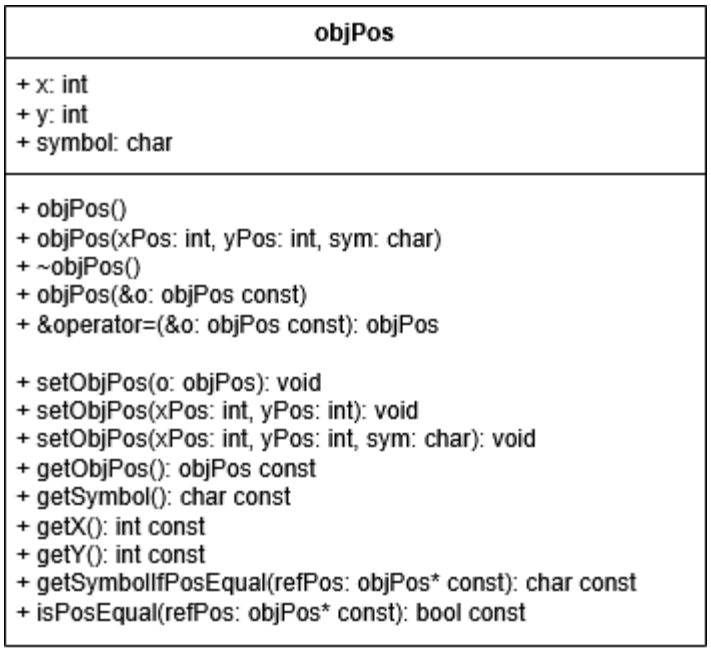
1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

   The design of the objPos class is not very sensible, as it provides an extra layer of abstraction around attributes `x` and `y` that does not present any advantages. Throughout the development process that we experienced, it seemed to only add extra confusion, having to access the x and y positions of an objPos by calling objPos.pos -> x or objPos.pos -> y, instead of being able to directly access the values.

   Moreover, the general purpose of using a struct like Pos is to group variables of similar natures. While it does succeed in doing so, its purpose is rendered obsolete by objPos, which offers the exact same functionality of grouping the variables x and y together, along with the symbol variable. In a class that effectively only has three variables, there is little benefit to additional groupings like Pos.

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. You are expected to facilitate the discussion with UML diagram(s)

   To improve the design of objPos, we would simply remove the use of the Pos struct, and have x and y be direct members of the objPos class. This would functionally act the exact same as the current objPos class design, but x and y become easier to access. For instance, instead of having to call objPos.pos -> x to extract the value of x from the Pos struct, you would simply call objPos.x. A UML diagram for this improved class design is included below.



It should be noted that in this class design, attributes x, y and symbol are kept as public variables for more convenient access. There is an argument for making these variables private, accessible only through the setters and getters provided. However, making these variables private would only be beneficial if the programmer wishes to add additional rules or constrictions to the values x, y and symbol may take on.