COMPENG 2SH4 Project – Peer Evaluation [30 Marks]

Your Team Members Maheer Huq Huzaifa Syed

Team Members Evaluated zhuk50 khans294

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.

In the main program loop of this team's code, we can clearly observe and interpret how the objects interact with each other. This may be because many of the objects declarations were done in the same manner as we did, making it easy for us to understand what was happening. For example, although we had different naming conventions, both of our teams had initialized a Player, GameMechs, and Food global pointer instances that were used throughout the main logic. It's clear that Player, GameMechs, and Food are all interacting classes as their instance of 'snake' is called by `Player(gamemechs, food)`. To add, we can easily tell how the objPos and objPosArrayList classes interact with each other by looking at a few lines in their code. We see that the Player class is tracking the positions of the snake body using objPosArrayList, where each individual segment is represented by an instance of ObjPos.

Overall, a positive feature about their object interaction is that you can map it to the given UML diagram clearly. We see what objects are interconnected by their calling in the code. There is not anything concerningly negative to note, however, we think they could have used more detailed names for their object instances.

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:

- Encapsulation; using OOD you can encapsulate different components of the game into classes (food, game mechanics, player position, etc.). This creates a modular feel for the code
- Reusability; classes can be easily reused and extended. If we wanted to add a new type of food like we had to for the bonus, we extend the Food class
- Simpler debugging; when you encounter an error surrounding a certain feature, for example a Player-related error, debugging is made simpler as you might expect to find your error within the Player.h or Player.cpp file
- Inheritance, Polymorphism, and Composition are all useful OOD techniques that can be used in our C++ implementation of the game

Cons:

- Lengthy setup; given that Snake is a relatively small game, the complex setup of all the classes might not be worth the effort and time in comparison to a brute force / procedural development style in C
- Performance; C tends to be slightly lower-level and closer to the machine code, this might render the non-OOD approach of the game to be faster if you are not careful

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 files where commenting is present, this team does a sufficient job of self-documenting coding style. This happens to be in their Player.cpp and Project.cpp files. In these two files they sufficiently explain any functions and complex logic through commenting. A small shortcoming of the commenting style is that they tend to leave one large comment at the top of each function, whereas it might be easier to break down comments per line where it makes sense to.

There are three classes that do not provide comments where it otherwise would have been helpful. These are Food.cpp, objPos.cpp, and objPosArrayList.cpp. These files each have at least one lengthy/complex function that would go well with comments, especially for anybody reading their code for the first time.

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.

For the most part, the indentation of the code is done well and logically. There seem to be little to no instances of lacking indents, which helps for readability. A shortcoming of the style of coding, however, is that in longer blocks of code where the logic becomes complex, the team could have used newline formatting and white spaces better. For example, in the DrawScreen(void) function in their Project.cpp, the entire logic behind drawing the game board gets very messy due to lack of spacing and even how the comments are placed.

A recommendation we have is to use empty lines whenever fitting, for example between conditional statements, for loops, and declarations of variables where it might make sense to. We also believe that placing comments above or below the corresponding line of code can help with the squished feeling, rather than placing comments adjacent to the already lengthy lines of code.

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 game is smooth and plays without any bugs; there aren't any missing features that are listed in the rubric. A small qualitative change we might suggest is to keep the game board upon losing the game so we can see how the snake collided with itself, as the current state of the game is for the board to instantly disappear. This, however, isn't an actual 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.

Yes there is considerable memory leak within this team's Snake Game. Upon running a drmemory report we see that there ranged to be between 3420-3452 bytes of leakage. We had expected some leakage as the team had` not implemented the minimum four for the objPos class, including the destructor. Every global pointer in the Project.cpp file was deleted accordingly, and all other destructors were implemented correctly. Hence, the root cause for the leakage appears to be the missing ~objPos() destructor.

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?

We believe the compound object design of objPos class is non-sensible, as the benefits are outweighed by the unnecessity and the cons. A benefit that could be made apparent is the organization of having an additional Pos struct, where clearly providing positional attributes as their own struct is improving readability. Despite this, the unnecessary nature of the struct Pos is just plain overkill. objPos only includes one extra attribute to the struct pos, so its unnecessary to even use the struct which just overcomplicates and adds to the code for no good reason. Also, the objPos has no real reason to be designed to have pos on the heap, which is usually typical for specific types of arrays which don't really fit the needs of 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.

I'd improve the object design by taking the suggestions above and implementing them. This is seen in the UML diagram below, where the Pos struct has been phased out and objPos is now very easy to understand. When taking the Pos struct out, the previous objPos attribute of pos is now replaced with x and y. Since the object is already labeled as objPos, it still makes sense to have an x and y directly as opposed to through a pos attribute, while also removing some of the clutter created by constantly having to write **objPos_name.pos->x**, and instead just writing **objPos_name.x**.