COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members dosanj5, yuehj

Team Members Evaluated mazumm4, deyn1

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.

   Yes, the interactions between objects in the main program loop can be interpreted relatively easily, as they are similar to how Dr.Athar's tutorials were structured. However, there are inconsistencies in the coding style, indicating a lack of communication between the teammates. Some parts have good coding practices (e.g., meaningful variable names and proper indentation), while others do not (e.g. have excessive pointer dereferencing, ambiguous variable names, violating Rule of Four Min six etc.)

   Here's a quick summary of the main positive/negative features we noticed:

- Descriptive Variable Names in Some Sections:
  - Variables like collided, currentY, currentY, boardX, and boardY in the Player.cpp file make the snake's death condition logic straightforward to understand.

- Issues with DrawScreen Implementation:
  - Lack of variables used create redundant accessing of values inside loops.
  - i.e. Looping over the entire game board multiple times to access object positions.
  - This causes slower runtime.
  - Overly long lines (i.e. if statement on line 86) make the code harder to read.

- Ambiguous Variable Names in Other Areas:
  - Names like game for a GameMechs pointer and valid for a flag are too generic and do not clearly convey their purpose.

- Violating Rule of Four minimum Six
  - **objPos** class is missing a destructor, copy constructor and copy assignment operator.

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:

   - Reduced global variables – Minimizes confusion and accidental misuse of shared data.
   - Reusability -- Classes enable code reuse through composition, resulting in cleaner, more modular code, and less clutter in the main file.
   - Scalability – Easier to add new features (e.g., player snake length, food generation) with minimal changes to existing code.

   Cons:

   - Increased complexity-- Managing relationships between objects (e.g., objPos and Player, Food and GameMechs) can become confusing.
   - Steeper learning curve -- Requires understanding advanced concepts like constructors, destructors, and the Rule of Four minimum Six, which demand more effort than procedural design.

<u>Peer Code Review: Code Quality</u>

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.

   There are little to no comments ~ (1-2) original comments per file. Comments were mostly from the skeleton code, which make it hard to understand what the code is doing, especially as mentioned before, there are a lack of descriptive variables used and methods from classes are directly referenced without any documentation as to what they are returning and why. To improve this code, I would add comments in the important sections such as the snake death condition in Player.cpp and the generateFood in gameMechs, to explain my logic.

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.

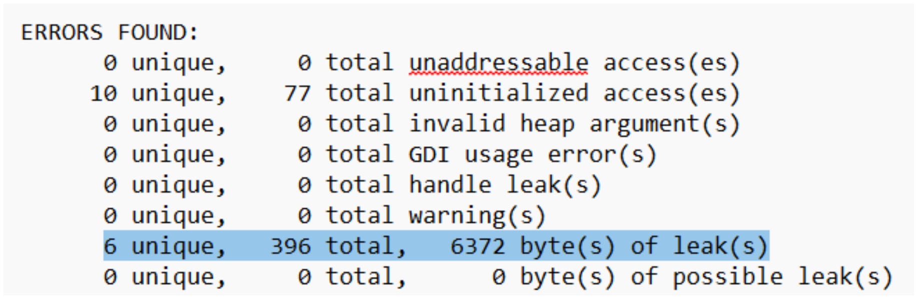
   The code generally follows proper indentation, but in certain sections, it is overly compact, making it difficult to interpret. For example, in the DrawScreen() function, the lack of whitespace between variable initialization (lines 67-68) and the subsequent for loops (line 69 and below) makes it hard to distinguish between different parts of the code. Additionally, the formatting of nested for loops is compact and challenging to read, as the curly brackets are placed in a condensed style. I would go with a more expanded and consistent bracket formatting, with additional spaces between nested blocks to improve readability. I would also add whitespace to separate sections and employ a consistent style for bracket formatting throughout the project. In contrast, methods like generateFood() in GameMechs are well-formatted, with clear spacing that separates initialization and logic (e.g., the spaces on lines 93 and 97), making it much easier to follow the flow of the 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)
   - Printing both exit messages on force exit; "You lost the game" and "You exited the game"
     - Because of the error mentioned below and the implementation of the exit message both messages would be printed.
     - Fixing the below bug would resolve this issue
   - Both exit features print for snake crashing into itself AND for force exiting the game.
     - ExitFlag and loseFlag are both set to be true when pressing space
     - Only set the exitflag to be true when pressing the space
     - Lose flag is properly set

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.

   This Snake game has 6372 bytes of memory leak as provided in the screen capture below.



```
ERRORS FOUND:
      0 unique,     0 total unaddressable access(es)
     10 unique,    77 total uninitialized access(es)
      0 unique,     0 total invalid heap argument(s)
      0 unique,     0 total GDI usage error(s)
      0 unique,     0 total handle leak(s)
      0 unique,     0 total warning(s)
      6 unique,   396 total,   6372 byte(s) of leak(s)
      0 unique,     0 total,      0 byte(s) of possible leak(s)
```

   Possible root causes:

   - **objPos.cpp** has no destructor (rule of four minimum six violated).
   - Memory allocated in **Player.cpp** (playerPosList) is not deleted in the destructor.

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?

Yes, it is sensible, for numerous reasons. The first being the use of encapsulation, the class grouped related data, x and y, together via a pos struct and the symbol separate but still within the same objPos class. This created a clear organization when it came to accessing the data in the object. The design also allowed for flexibility with its implementation, as in if additional positional data needed to be implemented later on that can be addded to the pos struct without the need to change the overall structure of the code. Also because the struct was implemented as a pointer, it allowed for dynamic memory allocation when needed making it an effective use of memory.

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(s).</u>

   One way would be to get rid of the pos struct and just have the variables be in private scope with a getter and setter as indicated below. This throws away all the benefits of encapsulation and resuability as provided by the pos struct. If the position data we want to store changes, we would need to change the class itself which violates the open closed principle of OOD. The design would also just be more messy and less modularized compared to how it is set like now

ObjPos

```
-   X
-   Y
-   Symbol
+   setX
+   getX
+   setY
+   getY
+   setSymbol
+   getSymbol
```