COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members ___Silver Owl: Zoe Pan (pan20-fall2024), Li Shi (shil41)_____

Team Members Evaluated ___best team ever: srikas15 & floraa1-fall2024_____

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.

   The main program loop is well-structured. The use of a while loop effectively controls the game's lifecycle by continuously calling the key functions (GetInput(), RunLogic(), DrawScreen(), and LoopDelay()) in this specific sequence, which made understanding of the flow of the game pretty easy. This sequence also ensures that there is a clear separation of concerns, where each component handles a specific aspect of game functionality. One possible improvement is in the use of the ! operator in the while loop condition. While this is a common practice, it may require readers to mentally flip the logic, especially if the function name (getExitFlagStatus()) doesn't convey the meaning clearly, the negation could make it slightly less intuitive. Instead, using while(myGM->getExitFlagStatus() == false) could make the exit condition clearer and more intuitive, especially for those who are not familiar with shorthand notations.

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:
   - Better organization: OOD separates the project into distinct classes (e.g., GameMechs, Player), making it easier to understand the overall structure compared to PPA3, where functions and global variables are mixed together in one big file.
   - Encapsulation and reusability: The OOD approach encapsulates data and functions within classes, which helps promoting reusability and making the code easier to maintain compared to PPA3, which lacks modularity.
   - Easy modification: OOD provides a more scalable design that can accommodate additional features and modifications more easily, while the procedural design in PPA3 can quickly become difficult to manage as the project grows.

   Cons:
   - Higher complexity: The OOD approach adds complexity as it deals with constructors, destructors, and pointers other than just regular functions, which was not present in the simpler C procedural approach of PPA3.
   - Tight coupling risks: If not designed properly, the relationships between classes in OOD can lead to tight coupling, where changes in one class affect others significantly. In PPA3, since the C procedural design relies on function-based programming without explicit relationships, it may avoid such issues.

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.

   The code includes sufficient helpful comments and follows a self-documenting style. Key parts of the code, such as DrawScreen() and major functions (GameMechs.cpp, Player.cpp etc.), have been annotated very well to describe their purpose and functionality. The naming conventions for functions and variables are also clear, making it easier to understand their roles within the code. The comments were concise and informative about the general role and mechanism of each function without being too abundant.

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 mostly follows great indentation practices, for example, the use of indentation in loops, conditionals, and functions helps visually differentiate between code blocks. Additionally, sufficient white spaces are used, which helps separate logical units, making it easier to read and follow the program flow. One of the two minor enhancements that could be made is improving the consistency of spacing around assignments and operations. For instance, in Project.cpp, lines such as:

   ```
   Line 66:  objPos playerPos = myPlayer -> getPlayerPos() -> getElement(0);
   Line 102:  myPlayer->movePlayer(); //move the player
   Line 125: objPos foodPos = gamemechs ->getFoodpos();
   ```

   Show inconsistent use of whitespace before and after the arrow operator (->). Ensuring uniformity in such spacing would enhance overall readability. Another minor enhancement would be adding MacUILib_clearScreen(void) at the beginning of CleanUp() as this can ensure that the game exits cleanly, without leaving behind any residual visuals on the screen.

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)

   An error is observed in line 111, in the DrawScreen() function of Project.cpp, the code suggests that a variable initialization problem after a failed attempt to compile the files. Specifically, the "variable-sized object may not be initialized" error in the DrawScreen() function is indicative of an attempt to declare an array (board[height][width]) with dimensions (height & width) that are not known until the game is initialized. However, arrays require fixed and compile-time constant. The problem may be solved by using dynamic allocation method such as creating a 2D array that can handle variable dimensions. This will resolve the issue with non-constant dimensions. A more efficient and straightforward approach, like the one our team used, involves directly printing the border characters while iterating through the board

dimensions without storing them in a separate array. Instead of creating an entire 2D array to represent the board, the drawing can be performed during the printing process using MacUILib_printf().

After the bug is fixed using my team's approach, the files can be compiled and the game runs smoothly with the correct implementation (correct eating and growing behaviour, forced end game method when the snake self-collides etc.) and proper presentation of the scores (grows each time a food is eaten) and end game messages.

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.

According to the memory profiling report, there is 1 leak of 16 total leaked bytes, traced back to the constructor of the Player class involving the allocation of an objPos object. This suggests that there is dynamically allocated memory that is not being properly released when the game ends.

```
Date/Time:       2024-12-04 01:30:49.428 -0500
Launch Time:     2024-12-04 01:30:47.188 -0500
OS Version:      macOS 14.6.1 (23G93)
Report Version:  7
Analysis Tool:   /usr/bin/leaks

Physical footprint:         2384K
Physical footprint (peak):  2384K
Idle exit:                  untracked
----

leaks Report Version: 3.0
Process 16598: 290 nodes malloced for 179 KB
Process 16598: 1 leak for 16 total leaked bytes.

Leak: 0x7fd240706850  size=16  zone: DefaultMallocZone_0x107c2a000   malloc in objPos::objPos()  C++  Project
        Call stack: 0x7ff806f50345 (dyld) start | 0x107c1d684 (Project) main  Project.cpp:40 | 0x107c1d735 (Project) Initialize()  Project
.cpp:63 | 0x107c1cfbd (Project) Player::Player(GameMechs*)  Player.cpp:33 | 0x107c1ceb4 (Project) Player::Player(GameMechs*)  Player.cpp:2
3 | 0x107c1c205 (Project) objPos::objPos()  objPos.cpp:9 | 0x107c1c1be (Project) objPos::objPos()  objPos.cpp:5 | 0x7ff80729634a (libc++ab
i.dylib) operator new(unsigned long) | 0x7ff8071237ea (libsystem_malloc.dylib) _malloc_zone_malloc_instrumented_or_legacy
```

The root cause of the memory leak in Player.cpp lies in the improper handling of dynamically allocated memory for the snake's segments, where memory is being allocated for the pos member in each objPos object using new without subsequently being deallocated. Specifically, within the Player constructor, memory is allocated as: segment.pos = new Pos; However, this dynamically allocated memory is never explicitly deallocated in the Player destructor. The destructor only deletes playerPosList: delete playerPosList; which removes the list but does not free the memory allocated for the pos pointers inside each objPos within playerPosList. As these pointers were allocated using new, they must be manually deleted to avoid memory leakage. To resolve the memory leak, the Player destructor should be updated to iterate through playerPosList and delete each pos pointer before deleting the list itself. This would ensure that all dynamically allocated memory for the snake's segments is properly freed, thereby preventing memory leaks.

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:

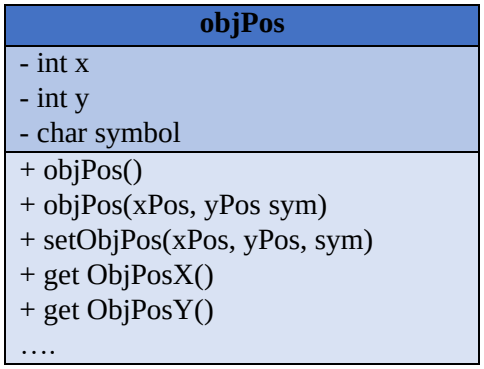
1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

Whether the compound design of the objPos class with the Pos struct is sensible essentially depends on what we're trying to achieve. On the positive side, having the Pos struct helps encapsulate the coordinates, which can make the code more readable when dealing with different positions. It adds a layer of abstraction, which could be useful if we need to frequently manipulate positions in a reusable way (for example, if we were extending the game to add more complex entities like obstacles, having Pos could make it easier to generalize movement and coordinate calculations.) This extra structure can make the logic a bit cleaner and make it easier to understand how positions are being managed.

However, this design also feels unnecessarily complex for what we're trying to accomplish with the snake game. The use of dynamic memory for the Pos struct adds more risk of memory management issues like leaks. It also makes the code harder to follow because of the multiple dereferences. Instead of directly storing x and y as simple members, having a separate Pos object seems to complicate things without adding a lot of value. If we had just kept x and y directly in objPos, it would have been simpler, with fewer memory concerns and less runtime overhead.

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>

An alternative design could focus on simplification and reducing memory management risks. Instead of using a compound object, the objPos class could be redesigned to directly store x and y as member variables like shown in the following UML diagram:



The improved design simpligies the representation of positional data for the snake game as the objPos class directly contains x and y as member variables instead of relying on a separate Pos struct. The code becomes easier to read and manage, and it removes unnecessary complications related to dynamic memory. This approach also helps ensure that the logic for the game remains more focused and less error-prone, so that it is more suitable for a lightweight game like this Snake Game created in this project.