COMPENG 2SH4 Project – Peer Evaluation

Your Team Members: Adam Idreis Mustafa Hassan

Team Members Evaluated: Vidanagamachchi Perera Nicholus Gossifidou

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions.

Part I: OOD Quality

1. **[6 marks]** OOD is about sensible code modularization. Looking at the header files of each object, can you easily interpret the possible behaviours of the objects involved in the program, and how they would interact with each other in the program? Comment on what you have observed, both positive and negative features.

   In the provided code, the header files (GameMechs.h, objPos.h, Player.h) serve as clear blueprints for the classes and their behaviors. These files encapsulate the classes, and the member functions within them exhibit a well-structured approach to modularizing the code. They have identical code structure to the UML provided in the lab instruction manual. This adherence to encapsulation ensures that member variables are accessible only through well-defined interfaces, promoting code maintainability. However, some member functions lack comments, making it less evident what their roles are. Adding comments to these functions would enhance code clarity. Furthermore, the header files could benefit from high-level descriptions of how these classes interact in the program to provide a holistic view of the design.

   In the main program logic, the code follows a standard game loop pattern, making it easy to understand the flow of the program. The interactions between objects, such as GameMechs and Player, are explicit in the logic. For instance, the main game loop orchestrates input handling, game state updates, and screen rendering by invoking functions from these objects. Nevertheless, there are sections of commented-out code (e.g., food generation) that may suggest incomplete features. These sections should either be implemented or removed to avoid confusion. Additionally, including comments to explain each section of the main loop's purpose would be beneficial, especially for code reviewers and future maintainers.

2. **[6 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 in project.cpp showcases both positive and negative features in its design. On the positive side, it embraces a structured program flow that includes distinct phases such as initialization (Initialize()), input handling (GetInput()), game logic (RunLogic()), screen drawing (DrawScreen()), and a delay (LoopDelay()). This well-organized structure enhances code readability and maintainability, making it easy to follow the flow of the game logic. For instance, the structured approach ensures that the game runs consistently until the exit condition is met, as indicated by myGM->getExitFlagStatus(). Additionally, the clear division of responsibilities between the GameMechs and Player objects is evident. The GameMechs object (myGM) appears to manage the game state, while the Player object (myPlayer) handles player-specific actions. This division of labor demonstrates a clear and logical object interaction approach, contributing to code organization.

However, there are certain negative aspects worth addressing. The main loop demonstrates tight coupling between the GameMechs and Player objects, leading to a direct dependency of Player on GameMechs for inputs and game state. This tight coupling can potentially hinder code flexibility and scalability. For example, the direct manipulation of myGM in Initialize() and GetInput() shows a strong connection between these objects. Additionally, the lack of abstraction in the main loop is evident, as it directly interacts with low-level details of the game objects. This could benefit from a more abstracted approach to simplify the main loop and enhance modularity. For instance, encapsulating actions within methods of a higher-level class could reduce the main loop's direct responsibility for these objects. Furthermore, the absence of error handling and robustness mechanisms is a concern. In a robust application, handling errors gracefully, such as invalid inputs or unexpected game state changes, is crucial for providing a smoother user experience. To further improve the codebase, considering design patterns suitable for game development (like State, Command, or Observer patterns) and introducing comments and documentation for clarity and maintenance would be valuable.

3. **[5 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 of C++ Compared to C:

1. **Object-Oriented Programming (OOP):** C++ supports OOP, allowing for better code organization and modular design through classes and objects.
2. **Abstraction:** C++ offers higher levels of abstraction, simplifying complex system handling and making code more expressive.
3. **Reusability:** With features like inheritance and polymorphism, C++ promotes code reusability, reducing redundant code.
4. **Standard Template Library (STL):** C++ includes the STL, providing a collection of powerful data structures and algorithms for efficient programming.
5. **Stronger Type Checking:** C++ enforces stricter type checking, reducing runtime errors and improving code safety.
6. **Exception Handling:** C++ supports exception handling, making it easier to deal with and recover from unexpected errors.
7. **Standardization:** C++ is a standardized language with well-defined specifications, ensuring portability and consistency across platforms.
8. **Scalability:** C++ OOD is produced on a bigger scaler for bigger projects rather than the C procedural design.

Cons of C++ Compared to C:

1. **Complexity:** C++ can be more complex, especially for beginners, due to its extensive features and concepts like classes and templates.

2. **Overhead:** C++ may introduce higher memory and processing overhead due to object management and runtime polymorphism.
3. **Learning Curve:** Learning C++ requires a solid understanding of OOP principles, which can be challenging for newcomers.
4. **Risk of Over-Engineering:** Developers may over-engineer solutions in C++, creating unnecessary complexity in projects.
5. **Compatibility:** C++ introduces some incompatibilities with C, making it harder to integrate C and C++ code seamlessly.
6. **Build Time:** Compilation and build times in C++ can be longer due to template instantiation and complex type resolution.

In summary, C++ offers advantages like OOP support, abstraction, and reusability, but it also comes with complexity, a steeper learning curve, and potential performance overhead compared to the simplicity and direct control of C.

Part II: Code Quality

1. **[5 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.

- **Comments**: The code lacks essential comments to explain its functionality, leading to reduced understandability. For instance, the purpose of critical functions and complex logic remains undocumented.
- **Self-Documenting Code**: While variable and method names are somewhat meaningful (e.g., **getBoardSizeX**), some could benefit from more descriptive names. The overall code structure follows a logical flow, enhancing readability.

  Improvement Suggestions:

  1. **Inline Comments**: Add inline comments to clarify intricate or non-intuitive code segments. For example, within functions with complex logic, such as **movePlayer**, comments could explain the decision-making process.
  2. **Function Documentation**: Each function should begin with a comment block that outlines its purpose, parameters, return value, and any side effects. For instance, the **Initialize** function should describe its initialization tasks.
  3. **Class-Level Comments**: Include comments at the start of each class file, providing an overview of the class's role and functionality. For instance, in the **Player** class file, explain its responsibilities related to the game's player.

To enhance the self-documentation of code, the team should focus on adopting the best practices such as employing more descriptive variable and method names for improved clarity. This involves replacing generic names like 'getInput' with specific alternatives like 'readPlayerInput' to convey the nature of the processed input. Consistency in naming conventions, whether using camelCase or snake_case, is crucial for uniformity throughout the codebase.

In addition to naming, general best practices play a vital role in enhancing code readability. This includes maintaining a clean code structure with proper indentation and spacing. Another strategy involves replacing magic numbers or characters with named constants or enums, exemplified by using 'Direction' instead of character-based directions. Moreover, modularizing complex functions by breaking them into smaller, dedicated units facilitates both readability and debugging. These practices collectively contribute to creating self-explanatory code that adheres to standards, ultimately improving the maintainability of the entire codebase.

2. **[4 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 maintains a high standard of readability through consistent indentation, appropriate spacing around operators, and logical newline formatting. Its indentation remains uniform across class methods and the main loop, enhancing code structure comprehension. Sensible spacing around operators and within control structures, such as if statements and switch cases, aids in code block differentiation. Newlines are effectively used to logically separate code sections, further improving clarity. To elevate code readability, ensuring consistent block style, managing line lengths, employing vertical spacing for logical grouping, aligning comments with code, maintaining clear indentation for continuation lines, and avoiding deep nesting are recommended. Overall, the code demonstrates commendable formatting practices, contributing to its readability and maintainability. Continual adherence to these practices and consideration of improvement suggestions will ensure ongoing code clarity, especially as the project evolves in complexity and size.

<u>Part III: Quick Functional Evaluation</u>

1. **[8 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 a technical user feedback)

1. **Player Movement and Boundary Conditions:**
    - Issue: The player's movement can be erratic, especially at the grid boundaries, where the snake can traverse over the borders incorrectly, leading to unexpected jumps and glitches.
    - Root Cause: Inaccurate handling of boundary conditions in the movePlayer method.
    - Debugging Approach: They should review the boundary condition logic to ensure that the player's position wraps correctly around the grid. Employ debugging tools such as breakpoints and logging to monitor position behavior near grid edges.
2. **Input Handling and Instruction Flickering:**
    - Issue: Input handling is suboptimal, resulting in occasional delays or unresponsiveness, impacting game responsiveness. Additionally, the instructions at the bottom of the screen flicker incessantly.
    - Root Cause: Input reading and processing in the getInput method may be blocking or inefficient, and rendering logic for the instructions in the DrawScreen function may require optimization.
    - Debugging Approach: Investigate the input polling mechanism to ensure it is non-blocking and efficient. Profiling tools can be useful for identifying input handling performance bottlenecks. Implement rendering optimizations for the instruction display to eliminate flickering.
3. **Food Generation and Exit Condition Handling:**
    - Issue: Food does not seem to generate in the game, and there appears to be no implementation for raising an exit flag to gracefully end the game.
    - Root Cause: Missing code for food generation and exit flag handling.
    - Debugging Approach: Examine and implement the necessary code for food generation within the game. Additionally, ensure that an exit flag is appropriately raised when the game should terminate gracefully.
4. **Memory Leaks and Resource Management:**
    - Issue: The game might suffer from memory leaks or inefficient resource management.
    - Root Cause: Potential mismanagement of dynamic memory, particularly concerning the player position list and other dynamically allocated resources.
    - Debugging Approach: Employ memory profiling tools to monitor memory usage and ensure proper management and release of dynamically allocated memory.

2. **[6 marks]** Does the Snake Game cause memory leak? If yes, provide a digest of the memory profiling report and identify the possible root cause of the memory leakage.

In the Snake Game code, there were potential memory leak concerns related to manual memory management using "new" and "delete." Specifically, dynamic memory allocation for player positions in the Player.cpp file and the allocation of GameMechs and Player objects in project.cpp raised red flags. However, upon closer examination, it appeared that the code did address these concerns by properly deallocating memory in the destructor of the Player class and implementing the CleanUp function for game objects. Additionally, the code did not show signs of exceptions interrupting memory deallocation. While vigilance in memory management is essential in C++, the provided code snippets did not exhibit memory leaks in the analyzed areas, thus ensuring memory integrity during gameplay.

Part IV: Your Own Collaboration Experience (Ungraded)

1. Tell us about your experience in your first collaborated software development through this project – what was working and what wasn't. If you are a one-person team, tell us what you think may work better if you had a second collaborator working with you.

In reflecting on our collaborative experience for this project, I believe there were some challenges in terms of workload distribution. It seemed that I took on the majority of the project's responsibilities, which at times felt unbalanced. While I understand that situations can arise that affect availability, it's important for both team members to contribute to the best of their abilities. Moving forward, it could be beneficial for us to establish clearer expectations and communication regarding project tasks and deadlines to ensure a more equitable division of work. -Adam Idreis