COMPENG 2SH4 Project – Peer Evaluation

Team Members: Seniru Perera and Nicholus Gossifidou
Team Members Evaluated: Mustafa Hassan and Adam Idreis

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.

Overall, the header files implement modularization and encapsulation well.

- Positives
  - GameMechs.h
    - The class is well commented, which provided clarity on the purpose and functionality of each group.
    - The private members are appropriately encapsulated to ensure data integrity and encapsulation.
    - The getter and setter methods are well defined, enhancing encapsulation of this class.
    - The behaviours of the objects are quite obvious, for example it can check the exit flag status, it can set the exit flag to true, it can check the lose flag status, and it can set the lose flag.
  - objPos.h
    - Also, correctly implemented.
  - Player.h
    - The private members are encapsulated, and appropriate getter and setter methods are provided.
    - The behaviour of the objects can be easily understood:
      - It can be instantiated with a reference to the game mechanics.
      - It can be destructed.
      - It can update its direction based on user input.
      - It can move itself.
      - It can get its position list.
      - It can set whether it has eaten food.
      - It can get its score.
      - It can check for self-collision.
- Negatives
  - The GameMechs class could use a destructor because it involves dynamic memory allocation for the objPosArrayList

- GameMechs class has both private and public members in the same section, which is fine overall, but could be organized more conventionally which allows for easier readability.
- hasEatenFood could also use a setter method

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 GameMechs and Player objects interact effectively, through methods like generateFood and getFoodPos. The code is also well-commented, which makes it easier to understand the purpose of these objects. We can easily interpret that the player class handles player-related functionalities, like updating direction, moving, checking collisions, while the GameMechs class manages game mechanics like generating food, and managing the game state.

However, we did notice some redundant conditions like 'if(myGM->getLoseFlagStatus() == false)'. That condition itseld is a Boolean, so it is false by default so it can just be replaced by 'if(!myGM->getLoseFlagStatus())'.

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:

- **Modularity**: using the OOD approach in C++ allows the project to be broken down into smaller parts, by separating different parts of the project into classes it makes the project overall more manageable.
- **Encapsulation**: Implementing the OOD approach allows for each class to have their own unique set of variables which cannot be accessed outside of the class. This encapsulation helps avoid unwanted interactions whereas in C procedural design the programmer must account for all interactions within the code.
- **Scalability:** OOD allows for larger scale projects as it is easier to manage and maintain than procedural design. Procedural design can get messy, and it can be hard to locate different parts of the code which may need to be referred to later in the project.

Cons:

- **Complexity**: OOD is not as simple as procedural design, implementation of pointers takes practice and a good understanding of OOD principles. There is a steep learning curve when starting OOD and it may be difficult to get the hang of things.

- **Refactoring challenges**: Since OOD requires separate classes that interact with each other based on the implementation, refactoring can prove difficult. Refactoring an object-oriented codebase can be more challenging, especially if the initial design was not well thought out. Changes in one part of the system can have ripple effects that are hard to predict due to the interdependence of objects.

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.

The code offers sufficient comments:

- Function comments
  - Exist for most of the functions in the header files, providing insights into the purpose of each function.
- Inline comments
  - Lots of inline comments explaining specific sections of the code, especially in the main program loop (project.cpp and player.cpp)
- Variable Naming
  - The chosen variable names are meaningful, and contribute to good self-documentation

Shortcomings:

- Detail
  - Some functions with complex logic, for example 'Player::movePlayer()', could benefit from more detailed comments, where the purpose might not be immediately obvious.
  - Furthermore, The input processing logic in Player::updatePlayerDir() could also be commented on in detail.
- Error Handling
  - Error handling isn't always commented explicitly. Adding comments to highlight possible error scenarios, or how exception cases are handled would be beneficial.
- Cross-File Comments
  - The code overall could use comments that explain the interaction/dependency between different classes throughout the whole project.

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 follows good indentation practices in general and includes white spaces.

Positives:

- Consistent Indentation
- Appropriate use of White Spaces
    - Used around operators, after commas, and in function calls.
- Newline Formatting
    - Used effectively to separate logical sections of the code.

Areas of Improvement

- Function Separation
    - Functions like Player.cpp are lengthy and would benefit from further separation into smaller functions, using white space/indentation.
- Vertical Spacing
    - A few places within the RunLogic function in project.cpp where extra vertical space between the logic would enhance readability.
- Consistent use of White Space
    - Could be more consistent with use of white space, especially after control flow keywords like if, else, switch. Would further enhance readability.

Part III: Quick Functional Evaluation

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)

After thoroughly testing the executable, it seems to work perfectly with no bugs. The implementation covered the basic requirements of the project, and all of the features work flawlessly. The board wrap-around is properly implemented as well as the food interaction process. The only critique we have is that more instructions should have been included when the program is run, such as directional keys for movement and the escape key. It is simple enough to figure out the controls based on assumptions however it would be more convenient to display these instructions on the screen, printed with the gameboard.

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.

After running *drmemory* there seems to be no memory leak within the project, this tells us that the dynamic memory allocation and deallocation have been implemented correctly. Each class where

dynamic memory allocation is used the proper measures have been taken to ensure that the memory allocated is cleared once the project is terminated.

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.

Working in parallel using the OOD approach was both efficient and effective, both of us were able to work on our own individual pieces of the overall project which in turn allows us to save time which can be used towards further refactoring and debugging. At times it also proved challenging as we were not always on the same page and the interdependence of classes and functions was difficult to achieve, while also ensuring there are no redundancies or unnecessary implementations within our project. Overall it was a good learning experience and provided plenty of insight.