COMPENG 2SH4 Project – Peer Evaluation

Your Team Members: Arya Pisharody, Peace Peace

Team Members Evaluated: Matthew Rozema, Noah Singh

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.
    - Looking at the header files of each object, it was quite easy to get the gist of what they should be doing and how they interact with one another
    - **Positive Features:**
        - When looking at the header file of an object, any other object that would be interacting with that object was included in the header file. For example, in the player constructor of the player header file, a reference was made to both the food and game mechanics objects because the player would be interacting with these objects. Additionally, the header files indicate what the behavior of the object might be. Some examples are in the Player.h header file, where it is evident the program to control the player's movement will be implemented in the player.cpp file, and the Food.h header file where the random food generation will take place within the Food.cpp file.
        - The layouts of all the header files were in a neat order that made it visually easy on the eyes and helped with the organization of the classes. Any interacting objects were placed at the very top of the header file, making it noticeable at first glance which other objects would be interacting.

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.
    - **Positive Features:**
        - All interacting objects were defined in the initialize function. This made It easy to see which objects would be referenced/used by other objects.
        - The clarity of the naming conventions for initialized objects facilitated easy comprehension; however, in scenarios involving interactions among multiple objects, intricate naming conventions could potentially confuse the reader. Nonetheless, the simplicity and appropriateness of the observed naming conventions ensured a straightforward connection between interacting objects. For example, in runLogic(), this group checked weather or not a self collision of the snake had occurred and if so would in turn raise the exit flag. From their

naming conventions of pPlayer and pGameMechs, it was clear that the player object and game mechanics object were interacting.

- Negative Features
  - There were instances where object functions should have been used but were not. As this project is built using the McMaster UI Library, and necessary print statements should have been done through MacUILib.printf(). While this group did use that on occasion, they also printed directly to the terminal which is bad practice. This occurred for their game instructions and their exit messages. In this case they used C++'s cout to print these messages.

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.
   - Overall, the C++ OOD approach was useful in the project as opposed to the C procedural design approach used in PPA3. Below, some pros and cons of the OOD and procedural programming approach are outlined:
   - **Pros of OOD:**
     - **Encapsulation** - encapsulation refers to the concept that some code and data that pertains to the implementation of an object should be "hidden" away from external objects. This aspect of C++ OOD allows the programmer to modify a certain aspect of the game, without affecting the other components. An example of its use in the code includes deciding whether the position the food should be generated is valid. The instructions outlined that food should not be generated and placed on the screen on top of the snake body. For this, a flag variable can be implemented to determine whether the randomly generated position is valid. Variables like this are used exclusively in the implementation of the Food object and have no reason to be visible to external elements.
     - **Abstraction** – abstraction is the idea that the focus of programmers will be how the objects interact with each other as opposed to the implementation of the object. This was helpful as the project was completed in a team as if one developer was responsible for the object's implementation, the other developer could simply use the objects and its various attributes to complete another aspect of the code.
   - **Cons of OOD:**
     - The C++ OOD approach can be unnecessary for smaller programs such as PPA3. In PPA3, there were significantly less "objects" (such as the player and food) to account for. To compare, PPA3 had a single character as the player, whereas in the final project, an array list of characters composed the player (and it kept growing). In addition to this, there was a fixed amount of randomly generated "food" in PPA3, but this is not the case in the final project. Using objects like in the OOD approach for a smaller project like PPA3 would be excessive.
   - **Pros of Procedural:**
     - For smaller projects such as PPA3, procedural programming can be more straightforward (and in some cases more efficient) as it does not involve making several objects to control various aspects of the code, as mentioned above.
   - **Cons of Procedural:**

- For larger scale projects (such as this one), the code can become disorganized, as it is harder to extend/modify. This was seen when developing PPA1, PPA2, and PPA3. Despite having several functions responsible for certain tasks, the code became increasingly disorganized and more difficult to add to as we progressed through the PPAs, and it relied very heavily on global variables. For a program like the final project, it is better to use an OOD approach to maintain organization in the code.

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 are provided wherever necessary, and for components that do not need comments, a self-documenting coding style is implemented. Sufficient comments for parts of the code such as variable declarations, if statements, switch statements, and for loops are provided. An example of where this is done well is in objPosArrayList.cpp in the "removeHead()" (line 45) and "insertHead()" (line 24) functions, where there is a brief comment explaining the block of code. An example in the code where this could be improved is in the objPosArrayList.cpp file on line 37 which has the comment "Full". It can be difficult for the reader to understand what this if statement is doing because the comment is vague. Additionally, there are not too many unnecessary comments in the program. For instance, in the objPosArrayList.cpp file, functions such as "removeTail()" (line 54) and "getHeadElement()" (line 59) are not commented, because it does not require comments (the name of the function tells you exactly what it does). Commenting such functions would be redundant.

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 is formatted well for better readability by including white spaces wherever relevant, and good indentation for the most part. To begin, the blocks of code are grouped together and separated from other blocks of code with white spaces. An example of a part of the code where this is done well is in the project.cpp file, where functions are separated from each other with white spaces (i.e. the "main()" function has white spaces before its declaration, and after its closing brackets). Another example includes initialization of related variables. Variables that are similar are grouped together (no white spaces in between), and separated from other unrelated variables. An example of this is the initialization of the global pointer variables in project.cpp. These are grouped together and separated from other "blocks" of code with white spaces. Similar to the placement of white spaces, the code mostly used good indentation to improve the code's readability. Examples of good indentation in the code can be seen wherever there are for loops, if statements, functions, constructors/destructors, getters, or other blocks of code enclosed in braces, the enclosed code is indented so the reader finds it easier to understand the code. An example in the code where this could be improved is in the project.cpp file on lines 107 and 108 where the for loop is unnecessarily indented.

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 running the program and playing multiple rounds, we did not encounter any noticeable bugs. We specifically checked if a "-" being eaten at just one snake body length would still subtract and result in no snake, and if the program would start lagging with an exceptionally long snake or at a high score (around 200 points was the highest we achieved).
    - Moreover, when the snake experienced self-collision, detected in the runlogic() function, the program promptly exited the loop without proceeding to the drawScreen() function. Other necessary requirements were also made, such as randomly generating food that did not spawn on the border, and that the snake wraparound did not also collide with the border.
    - In summary, the program met all requirements, demonstrated user-friendly features such as reasonable game speed, and included a helpful menu.

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.
    - No memory leaks were identified, as such a digest of the memory profiling report cannot be provided.

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.
    - What worked:
        - Using GitHub to push/pull updated code worked well. This allowed us to work efficiently as it clearly outlines the changes each person has made to the files
        - Splitting the work into Developer 1 and Developer 2, where each partner works with parts of the code that do not necessarily interact with each other, then combining the code so the objects each developer worked on can interact with each other.
    - Things to Improve:
        - As we write code, we should add comments explaining what that segment of code does. This will allow for the partner to debug/troubleshoot the code more effectively and could save time when trying to fix an error.