COMPENG 2SH4 Project – Peer Evaluation

Your Team Members	Alexander Arruda	Bradley White
Team Members Evaluated	Mehak Shah	Nandha Dileep

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.

   The possible behaviours of each object can easily be interpreted from the header files. It is clear that the "Food" object will take care of food behaviors and that the "Player" object takes care of player information and tracking, for example.

   In addition, each header file has methods with descriptive names, sensible return types, and input parameters. For example, in Player.h, "checkFoodComsumption()" and "checkSelfCollision()" both have very clear names in indicating their function, and they are both defined to return type bool, which seems sensible given they are "checking" functions. Also, the input parameters of these functions are empty, supporting that they are used to check information and simply return a bool. Also, the method names do not indicate that they will modify anything, and so the return types and parameters support this.

   Furthermore, looking at the header file, based on the included parameters and methods, it is clear how each object interacts with one another. For example, it is clear that the "Player" object, will interact with both the Game Mechanics and the Food objects, since the player header file has parameters to hold references to instances of these objects. The Player object also has a "checkFoodComsumption()" method in its header file. The method name clearly indicates that it will be interacting with the Food object within that method. On the other hand, in the "Food.h" header file, "generateFood(objPosArrayList &playerPosList)" makes it clear by its arguments that it will interact with the player object, hence &playerPosList.

   This said, while the names and arguments of most of the methods make it clear what they do and how they interact with other objects, some do not. For instance, the movePlayer() function in the Player header file: this name makes no indication that this function will be responsible for also checking if the new player position collides with food—which is does. Also, increasePlayerLength() does not make it clear that this is the function responsible for generating new food (by calling generateFood() within it).

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.

   By observing the main logic in the main program loop, it is not always clear how all objects interact with each other. For example, in the Run Logic section, the main program simply calls:

   ```
   myPlayer->updatePlayerDir();

   myPlayer->movePlayer();
   ```

   This hides the fact that to move the player, collisions with itself and food items are checked, border wrapping-around is checked, and the entire generating new food process is called. For the first two features, this is OK, since the collision detecting and wrap-around processes are expected by the nature of the updatePlayerDir() and movePlayer() names, but generating new food is not inherently expected to be carried out within these functions.

   This is confusing to someone who does not know how the Food, GameMechs, and Player classes were designed to interact. They will have no idea why there is nowhere in the main logic to generate the new food.

   This said, to someone who is familiar with the design of these classes, and how to properly link them, this abstraction from all the underlying processes makes the main code much simpler.

   All this said, the abstraction from the underlying mechanisms is a positive of taking this object-oriented design approach. However, in this one instance, the abstraction hides too much, making the code un-interpretable; an entire process *seems* to be missing, but really is there.

   On the other hand, it is explicitly clear in the DrawScreen routine how the objects interact with the main code, since the GameMech's parameters are used to determine where borders should be printed, and the player and food classes are used to determine where and how to print them. The implementation is written clearly and is obvious to the programmer, and thus a positive feature.

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:

   - Less global variables (less namespace contamination).
   - Modular: reusable and easier to debug as-you-go (section by section).
     - Encourages incremental engineering in ensuring each object is complete before proceeding.
   - More organization: all related variables and functions grouped together. Leads to ease of understanding compared to procedural programming.
   - Abstraction in the main program logic, makes code simpler to implement and read.
     - Easier to check-list intended functions of program.

   Cons:

- Can lead to too many methods being created for simple procedural functions; more time, effort, and storage consumed.
- To keep modularity, more objects would need to be created, thus bulking up file size and complicating code referencing.
    - Larger file sizes can decrease performance.
- Procedural design allows for **all** functions to be explicitly obvious and present at once in the main file, whereas OOD hides operations in objects.
- Retroactive debugging becomes more difficult due to potential errors being present across different files needing to reference and use variables/pointers in different ways.
    - i.e., debugging after not following incremental engineering

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 comments present in the code are sufficient in clearly summarizing what each section of the code does. Each major section is introduced with a comment to summarize the main function of that section, with further comments summarizing key components inside. For example, in the "DrawScreen(void)" function in Project.cpp, each loop, statement, and variable assignment is introduced with a comment clearly summarizing what that section is supposed to do, such as draw border characters, check if the player needs to be drawn, or printing food pieces.

   However, in some instances, the comments do not clearly describe the logic behind the operations at hand, which can even be confusing to a returning programmer trying to understand unobvious decisions. One case of this is within Player.cpp's "increasePlayerLength()" function. The comments clearly describe the operations of what is done in the code. However, this function also generates food, which is not an obvious operation to be performed when interpreting the purpose based on the function name. The code describes that it *does* generate food, but not *why*. Though this is not always necessary, explaining this decision would allow for programmers to better understand the logic behind the design, as well as better employ a sufficient self-documenting coding style. This could be improved by incorporating comments to explain why decisions like this were done. This would help new readers and the returning programmer when they forget the exact logic behind their design.

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 mostly follows good indentation, with all contents of functions and loops indented to indicate they belong to that function/loop. There is also an appropriate use of newlines, where they utilize a new line between smaller code blocks withing each function to indicate that they perform different sub-functions. This said, there are also times when there may be too much whitespace characters, especially in the case of simple switch cases or if/else statements with

only one statement. For instance, the updatePlayerDir() method has a switch case that has if statements within it, presented as:

```
Switch (case)
{
        Case 1:
                If (condition)
                {
                        statement;
                }
                Break;
        Case 2:
                If (condition)
                {
                        statement;
                }
                Break;
}
```

This is good practice in all ordinary cases for organization. However, since the if statements only have single sub-statements and are nested within a switch case, this visually adds a lot of space, making it overall more difficult to read and navigate. I would suggest in cases like this, with very simple if statements, to format the code as such:

```
Switch (condition)
{
        Case 1:
                If (condition) {statement;}
                Break;
        Case 2:
                If (condition) {statement;}
                Break;
}
```

This, just helps to condense the code, making it briefer and thus a little easier to read and work with.

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)

   The Snake Game created by the evaluated team offers a mostly bug-free and smooth experience. There are only 2 visual bugs, but upon inspection these bugs have the same origin. These buggy

features are the fact that (1) the snake does not seem to instantaneously grow when eating a food, and that (2) the snake appears to freeze for a few frames when eating a food. Note that that despite these bugs the score increases instantaneously when eating a food, indicating the error probably lies specifically in the increasing length or collision detection features. Specifically, the issue probably lies in the process of adding a new head to the snake; in the increasePlayerLength() function, an insertHead function is called. This, logically, should add a new piece to the head of the array. However, since there is no shift or size increase when the food is immediately eaten, the error may lie within the insertHead logic, or how it is utilized within the increasePlayerLength() function. This said, since the snake does eventually grow, this indicates that there may be errors in the coordinates of the new location of the head as well.

For the evaluated team, it would be recommended to first execute their code and play the Snake Game thoroughly, testing possible cases where there may be errors. Slowing down the speed of the game would further help analysis reveal obvious issues. Using a debugger, such as GDB, could be used to ensure that snake piece coordinates at different frames of the game are as they should be, i.e., checking to see that the added head's coordinates are as expected, focusing on fixing it the debugger reveals otherwise. More generally, further analysis of the specific sections of code where these errors may be present, and doing a manual, on-paper analysis of the code would help the programmers better understand the logic and see where mistakes may appear. Lastly, ensuring that all parameters passed are what is desired is crucial to functional code; they should make sure that they are passing into functions what they want to pass in, and understand what is available at different sections of the code and when global variables are updated. Overall, a focus on incremental engineering as early on as possible will allow for the programmers to be able to tackle errors as soon as possible.

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.

   Dr Memory indicates that the Snake Game does not cause memory leak. Inspecting the code visually, we can also see that all heap instances of objects in the main program are being properly deleted. All heap members are deleted where necessary, such as in the CleanUp routine in Project.cpp, in each destructor for the Player and objPosArrayList objects (the only objects in their project that allocate heap members), and within functions that resize previous allocations.

   In the Project.cpp file, global instances of pointers to heap objects are created. These are appropriately all deleted in the CleanUp routine, which is called along the program and thus prevents leakage.

   The Player and objPosArrayList objects both have destructors which delete their respective heap members. GameMechs and Food have been left to have their default destructors, which is good practice since neither of those objects allocate heap members, and so no deletion is necessary.

   Finally, for good memory practices, they have even added a safety measure of deallocating the current array in the objPosArrayList class and reallocating a new one with an increased maximum heap storage, for when the list (in this case the player length) begins to exceed the

predefined 200 maximum capacity. This feature is a good practice, ensuring that the program will never encounter a segmentation failure, even when dedicated players might reach scores beyond 200 points.

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.

   For a first collaboration software development experience, it went smoothly. The OOD nature of the project allowed for easier collaboration. One aspect that was confusing initially was best utilizing the Git pull and push functions, as when working individually on the same parts, each of us did not want to accidentally override each other's code. When we each had code created for new sections that depended on different components and had to be rewritten, efficiently reworking the code and deciding between each other's logic was a challenge.

   Another challenge was making sure that we each understood each other's code once we joined back together to complete the implementation of the main program, and the interactions between objects. Our understanding of the logic of code is both different, and so each of us had different design implementations and ideas in our minds that were at times difficult to translate to each other (pen and paper often being required to get our ideas across visually).

   Another thing that worked well was setting deadlines, meetings, and having clear communication. This ensured we worked to keep up with the timeline of the due date and were able to sit together and focus as we worked, not having to meet impromptu and thus not be fully engaged. The deadlines also enforced us to communicate clearly for when we felt we needed to move deadlines or would not meet them. It encouraged and nurtured an environment where we were both aware the progress of the project and working towards the completion of it diligently and eagerly.

   Nonetheless, the collaboration went very well, and we were both able to work together efficiently and put both of our minds together to design the project. By the time we were working on the bonus, we were a well-oiled machine, enjoying the process of designing our own functions.