COMPENG 2SH4 Project - Peer Evaluation

Your Team Members: Kendall Bird and Megan Saunders
Team Members Evaluated: Yax Patel, Sharvin Soosiapillia

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.

Firstly looking at the **Food** class, the objects are well-organized and prototyped appropriately. The default and copy constructors *Food(GameMechs *thisGMRef)* and *Food(const Food &food)* are identified with comments and grouped together with the *~Food()* destructor, making the header file organized. The other functions defined in the class are named appropriately so a user can infer what the implementation aims to achieve. For example, the void function *generateFood* checks for appropriate spaces to generate a random position on the gameboard for a random food type. The **GameMechs** class is also well-organized and prototypes are named appropriately, but there are no comments in the header file as shown to the right:
A few short comments would be helpful for a user.

```
class GameMechs
{
private:
    char input;
    bool exitFlag;
    bool loseFlag;

    int score;

    int boardSizeX;
    int boardSizeY;

public:
    GameMechs();
    GameMechs(int boardX, int boardY);

    bool getExitFlagStatus();
    bool getLoseFlagStatus();
    void setExitTrue();
    void setLoseFlag();

    char getInput();
    void setInput(char this_input);
    void clearInput();
    void incrementScore();
    void incrementScore(int count);
    int getScore();

    int getBoardSizeX();
    int getBoardSizeY();
};
```

Moving on to the **Player** class, all the function & object prototypes are mostly grouped together based on their purpose, and are named appropriately. The references to other classes *GameMechs *mainGameMechsRef, Food *foodRef* and *objPosArrayList *playerposList* are all labeled with comments and clearly show the interaction of this class with the other classes in the project - although the *objPosArrayList* reference is not located in the same place as the other references, which may cause users to miss it or get confused. Lastly, the **objPosArrayList** class is referenced by all the other classes, and all the included functions are named appropriately. However the private data members in this class do not have any comments associated with them, and because they are all named similarly (*objPos *aList, int sizeList, int sizeArray)*, it is confusing to users what the purpose of each of them is and how they differ. Choosing a more descriptive identifier or adding comments would fix this.

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 *Project.cpp* file is generally well organized and logical. Pointers to each of the necessary GameMechs, Player & Food classes are included first in the global variable definition, which makes it clear that this code will reference objects from these classes.The *DrawScreen* function is quite long and looks confusing at first glance, but as variables and functions have been named appropriately and consistently through the other classes, lines such as:

```
player->getPlayerPos(pos);

player->getPlayerPosList();

food->generateFood(player->getPlayerPosList());
```

Can easily be identified as retrieving the initial position(s) of the player object and generating an initial food item on the board using the player's positions as input so as to not overlap with the snake body.

However, there are a few odd features in the void *RunLogic(void)* function. Firstly the function includes the following expected logic;
player->updatePlayerDir();
player->movePlayer();
These two lines firstly call on the function to update the player's direction based on the input, then move the player in the updated direction. These functions are from the Player class, and it is easy to follow this logic back to the Player class to inspect the implementation. After these functions, the following loop is included:

```
if (game->getInput() != 0)
{
    if (game->getInput() == 'l')
    {
        game->setExitTrue();
    }

    if (game->getInput() == 'i')
    {
        game->incrementScore();
    }

    if (game->getInput() == 'r')
    {
        food->generateFood(player->getPlayerPosList());

        objPos foodPos;
        food->getFoodPos(foodPos);
    }
}
```

This loop is unexpected in the main RunLogic loop, as it allows the user to "break" the game. Inputting "l" ends the game, despite indications that the main exit input key is "space". Inputting

“i” increments the score by 1, and inputting “r” generates 5 foods in different random positions on the board. Although these features don’t ruin a user’s interpretation of the code, they are confusing and should be removed. These were probably included as a way for the designers to test their implemented functions easily, and are very useful in that way but should not be accessible by a user.

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.

C++ OOD Pros

- Classes such as GameMechs, Player and Food encapsulate related objects and functions, allowing for code to be more organized and modular
- It is much easier to reuse functions included in different classes by creating references, (like *GameMechs *game* in the Project file) making it easier to expand upon in future code
- Objects and functions are organized within classes, reducing the amount of global variables needed and preventing data being accessed in inappropriate locations

C++ OOD Cons

- As each class relies on each other, (in particular the classes *objPos* and *objPosArrayList*) are referenced by every other class) it can be much harder to debug issues when you aren’t sure if the problem is in the class that’s throwing the errors or in one of the classes it is referencing
- In general, the program lengths are much longer than procedural programs, and the amount of referencing that needs to be done makes the code more complex

C Procedural Pros

- Procedural code is generally more straightforward, and users would not have to navigate many different files in order to connect different functions with their implementations
- Designers can get right into coding implementation, rather than setting up class structures and different files for organizational purposes. The PPA’s took much less time than the main project because of this

C Procedural Cons

- In PPA3, much more global variable definition was required. Excess global variable definition could lead to a number of issues, especially aspects of a program being modified globally when they should only be modified locally. OOD approach greatly reduces this
    - The only global variables needing to be defined in the main Project.cpp file were 3 references to other classes, whereas the PPA3 assignment could contain up to 9 global variables depending on individual implementation
- For larger projects that contain many iterations such as our Snake game, it might be harder to deal with the project growth as procedural programming is done in one big module

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.

In general, the code was well commented throughout the classes and functions to increase readability and understanding of the code. There was a lack of comments in the header files, however, which would make the purpose of the code easier to follow. One instance where additional comments would have made the code more effective was the implementation of **DrawScreen** in the **Project** class. The ASCII values of the characters used were listed, but since they varied from the traditional project requirements, it would have been helpful to have a comment beside each number that specified the character being drawn.

Also in the *Player.h* header file, some unnecessary comments were left in such as *"upgrade this in iteration 3"* and *"You will include more data members and member functions to complete your design"*. These comments would have been useful for the designers while they were creating the code, but are confusing for users when they are included in the completed code. These comments could simply be removed.

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.

Throughout all the files in the project the code followed correct indentation practices, white spaces and newline formatting which increased the code readability.
Here is an example of the code in the DrawScreen() function that shows correct formatting:

```
for (int i = 0; i < game->getBoardSizeY(); i++)
{
    for (int j = 0; j < game->getBoardSizeX(); j++)
    {
        // if border
        if (i == 0 || i == game->getBoardSizeY() - 1 || j == 0 || j == game->getBoardSizeX() - 1)
        {
            // top left corner
            if (i == 0 && j == 0)
            {
                MacUILib_printf("%c", (char)201);
            }
            // top right corner
            else if (i == 0 && j == game->getBoardSizeX() - 1)
            {
                MacUILib_printf("%c", (char)187);
            }
```

This could be a very confusing function to look at if proper indentation was not used, and the addition of comments to separate different points in the loop makes it very visually easy to follow. This formatting style was consistent throughout all the files, which made the code easy to follow as well as feel consistent and polished.

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 ran very smoothly for a bug-free playing experience. While playing the game, the snake array updated very well and almost immediately after collecting a food item on the board. Additionally, the various food items that could be picked up (X, = and +) each interacted seamlessly with the snake head as it collected the items, despite the different game updates they caused. The border wraparound implementation in the DrawScreen() function of the Project class was perfect as well, resulting in the snake transitioning from one side of the screen to the other without any lagging.

The implementation of the various functions to alter the array list for snake movement was effectively done as well. The *insertHead(objPos thisPos), insertTail(objPos thisPos), removeHead(), removeTail()* were all implemented efficiently and effectively which added to the smooth playing experience. The extra display the team added at the top of the terminal; *"WASD to move, space to exit"*, was a useful addition that makes the game suitable for players who are not familiar with the functionality of the game.

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.

The Snake Game did not cause any memory leak (0 bytes as per the Dr. Memory report we generated). This is because the designers used delete at the end of the program for the instances where they allocated memory on the heap.

In the Food.cpp class, the implementation of the Food deconstructor: Food::~Food() contains the necessary 'delete' statement to free the dynamically allocated memory. The corresponding header file Food.h includes the deconstructor which also shows the correct memory deallocation in the Snake Game code. The second instance where a deconstructor was correctly implemented in their code was in the objPosArrayList class. In this case, the deconstructor correctly uses "delete[]" to free the dynamically allocated memory for the array of objPos objects. The use of delete[] in the destructor of objPosArrayList is related to the fact that the memory for aList was allocated using new objPos[sizeArray] with square brackets, indicating an array of objPos objects. As was the case in Food.h, objPosArrayList.h also includes the deconstructor. Therefore, the implementation of the code is congruent with the memory profiling report showing a leak-free game.

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.**

We had a very successful experience working collaboratively over the course of this project! Working in parallel following the workflow worked best for us, and we met often to discuss our implementation when we were putting our parts together. The main learning curve we experienced was learning how to branch and git pull each other's code properly and without overwriting our own work, as we hadn't done it prior to the project.