COMPENG 2SH4 Project – Peer Evaluation [25 Marks]

Your Team Members: stoicap, harria49

Team Members Evaluated: khawam9, kaurs89

Provide your genuine and engineeringly verifiable feedback. Ungrounded claims will lead to deductions. Completing the peer code evaluation on time will earn your team a total of **25 marks**. Do not exceed 2 paragraphs per question.

Peer Code Review: OOD Quality

1. **[3 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.

It is easy to interpret how the objects interact with each other, partially because the interactions are well-described, both through the names of methods and fields, and through appropriate commenting. One positive feature was the decision to store commonly called values (e.g. the dimensions of the board, position of the food and player) as local variables to eliminate unnecessary function calls and improve the readability of the code. In addition, the design mostly manages to encapsulate important functions and data into classes. At some points, however, the designers should have paid more attention to encapsulation. For example, lines 64-67 of Project.cpp check whether the input is SPACE in order to stop the program. This is questionable because the Player class already contains a method to make decisions based on user input, and that method is called just prior, at line 60. This exit check should have been included in there. OOD provides the opportunity to group related functions and data together, and here the programmers have rejected that opportunity for no apparent reason.

It is important to discuss that the design is not very abstract, as there are very few new methods or classes other than those suggested by the project. There are advantages to this, for example it makes the logic easier to understand from basic programming principles. However, the addition of new methods for common tasks would have contributed to modularity and made it easier to expand the code. For example, the program often iterates through an objPosArrayList and checks whether any contained objPos objects contain a certain pair of coordinates. This happens in the main loop (Project.cpp) at line 108, and is also essential to checking for player death and proper food placement in the Player and GameMechs classes. It would continue to be important if, for example, the developers wanted to have a food bucket instead of just one food item. This is an example of a routine communication between two objects, so it makes sense to introduce a method for the objPosArrayList class to handle it compactly under the hood. It is instead done procedurally every time and that is a missed opportunity.

2. **[3 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.

Benefits of OOD:

- Modularity: enables bottom-up approach involving well-defined building blocks
- Encapsulation: only relevant details are exposed, simplifying the main program logic
- Organizes and protects information, which is good for teamwork
- Code is reusable and expandable, so it's easy to add new features e.g. a food bucket
    - In particular, objPos and the array list are extremely versatile for tracking many related objects

Drawbacks of OOD:

- Risk of overabstraction: use of classes can make the logic less intuitive than procedural programming in simple cases such as the PPAs
- Code is long with many calls to keep track of
- Difficult to change/refactor code in many files with many interactions
- Difficult to debug because of many definitions in many files

Peer Code Review: Code Quality

1. **[3 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 coding style is self-documenting, with descriptive and concise variable names. There is very little hard coding that makes the code confusing and unreadable. Where long, unruly conditional statements are found, appropriate comments are made to clarify the function of the statement. In some places, comments are sparse, but the easy-to-read nature of the code makes them unnecessary.

If there is one place that could be better documented, it would be the Player class, which has some confusing object declarations. For example, from lines 90-153, the code is not sufficiently self-documented. currentHead is instantiated with no explanation, modified across a long switch statement, and eventually inserted. Somebody who has not already done the PPAs might have some difficulty navigating the code because it relies on prior knowledge. currentHead is also not a great variable name because its purpose is to eventually act as the new head to be inserted, at which point it is no longer the current head. In areas like this, a high-level explanation of the purpose of each design choice (e.g. decide where to put the new head based on current direction) would provide context for the code, making it easier to read.

2. **[3 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, separating appropriate parts of functions and giving the code an understandable structure. For example, the overwhelming majority of loops employ indentation even if they contain only one line of code, giving the reader a good sense of hierarchy and scope. There is plenty of white space in most of the classes (e.g. the switch case in the player class) which separates the code into easily digestible islands that make it easier to read.

   One notable exception is the objPosArrayList.cpp file, where a lot of object declarations and heap commands are packed somewhat densely. We could improve it, for example, by inserting whitespace before the for loops at lines 27, 71, and 96.

Peer Code Review: Quick Functional Evaluation

1. **[3 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 technical user feedback)

   Wraparound logic, directional changes, and food eating are free of bugs. Food generation occurs at the appropriate time and is always successful, but can sometimes result in food being generated inside the body of the snake, as shown in the figure below. The obvious place to look to solve this would be the generateFood logic in GameMechs.cpp.

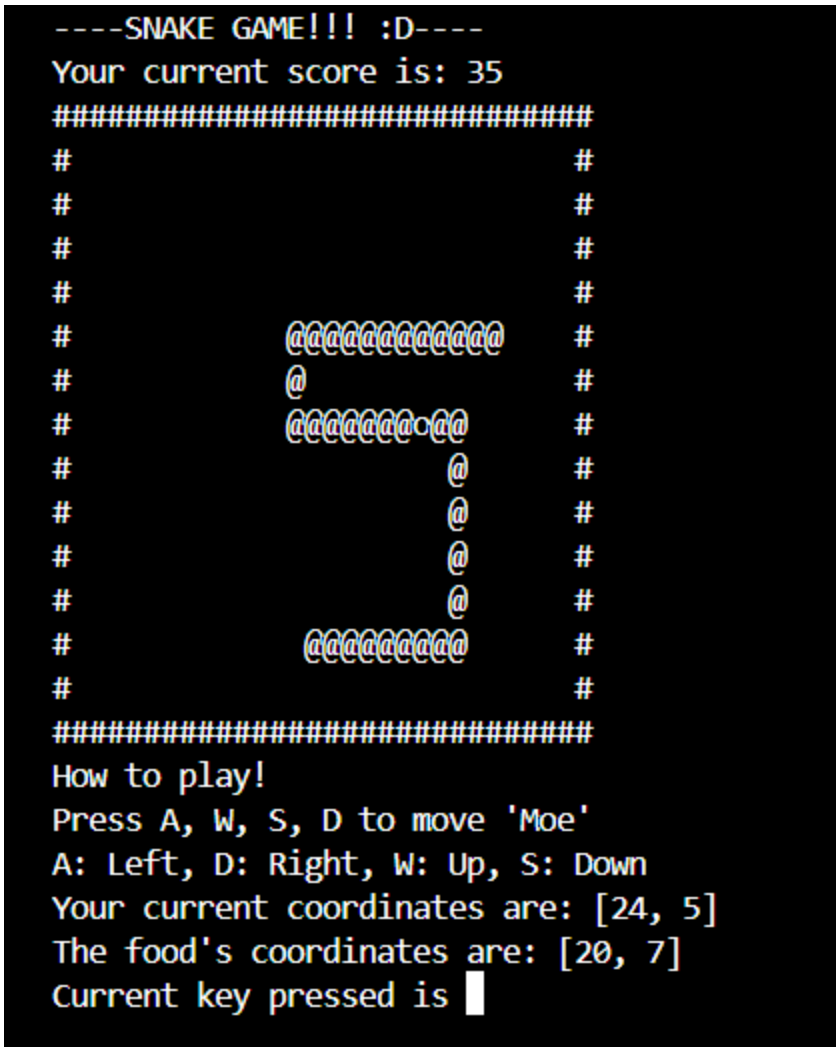

Figure 1. The food can incorrectly spawn and remain inside the snake body.

The debugger is arguably not necessary. Including a statement to print the value of isValidPos would reveal that it is true in almost all cases. The reason for this is ostensibly that isValidPos is set to true (line 143) if *any* element of blockOff does not share its position with foodPos (line 141), which is always true if the snake has length greater than 1. This guarantees that foodPos will land in the first randomly generated place almost all the time. If even one element of blockOff shares its position with foodPos, it should guarantee failure, but instead the program prematurely guarantees success.

2. **[3 marks]** Does the Snake Game cause memory leak? If yes, provide a digest of the memory profiling report and identify the possible root cause(s) of the memory leakage.

   After using the DrMemory tool, it is clear that this snake game does not cause a memory leak. There was a memory leak of 2 bytes, but it was clearly not the fault of the snake game itself. Rather, some other files or systems associated with the project caused a leak to appear in DrMemory.

```
Error #31: LEAK 2 direct bytes 0x01a00c90-0x01a00c92 + 0 indirect bytes
# 0 replace_malloc                              [D:\a\drmemory\drmemory\common\alloc_replace.c:2580]
# 1 msvcrt.dll!_strdup
# 2 .text                                       [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/MacUILib.cpp:47]
# 3 __mingw_glob                                [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/MacUILib.cpp:47]
# 4 _setargv                                    [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/MacUILib.cpp:47]
# 5 .text
# 6 mainCRTStartup
# 7 ntdll.dll!RtlInitializeExceptionChain    +0x6a     (0x77d5c0cb <ntdll.dll+0x6c0cb>)
# 8 ntdll.dll!RtlClearBits                   +0xbe     (0x77d5c04f <ntdll.dll+0x6c04f>)
```

Figure 2. The sole memory leak in the report.

Tangential to this question is the question of heap errors, of which DrMemory reports no less than 14 unique cases. In each of these cases, DrMemory complains that memory was allocated with new and freed with delete[]. Without going too deeply into this tangent we can say that the other team should use delete and not delete[] in the objPos destructor, since objPos is not an array.

```
Error #17: INVALID HEAP ARGUMENT: allocated with operator new, freed with operator delete[]
# 0 replace_operator_delete_array_nothrow              [D:\a\drmemory\drmemory\common\alloc_replace.c:3002]
# 1 objPos::~objPos                                    [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/objPos.cpp:51]
# 2 objPosArrayList::~objPosArrayList                  [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/objPosArrayList.cpp:56]
# 3 Player::~Player                                    [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Player.cpp:21]
# 4 CleanUp                                            [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Project.cpp:156]
# 5 main                                               [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Project.cpp:38]
Note: @0:00:15.473 in thread 28952
Note: memory was allocated here:
Note: # 0 replace_operator_new                          [D:\a\drmemory\drmemory\common\alloc_replace.c:2903]
Note: # 1 objPos::objPos                                [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/objPos.cpp:5]
Note: # 2 objPosArrayList::objPosArrayList              [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/objPosArrayList.cpp:17]
Note: # 3 Player::Player                                [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Player.cpp:12]
Note: # 4 Initialize                                   [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Project.cpp:47]
Note: # 5 main                                         [C:\Users\fujid\OneDrive\Desktop\COE2SH4\Project Code Review\course-project-heapnotheap/Project.cpp:28]
```

Figure 3. The last heap error in the report.

Project Reflection

Recall the unusual objPos class design with the additional Pos struct. After reviewing the other team's implementation in addition to your own, reflect on the following questions:

1. **[3 marks]** Do you think the compound object design of objPos class is sensible? Why or why not?

No, it is not sensible. Every time the program accesses the x and y coordinates of an object, it must dereference pos and make an extra call. The manual justifies this by calling it a 'memory conserving strategy', but it is unclear whether that is worth the impact these extra calls have on runtime.
The calls happen very often in the other team's program, at least 20 times per second for each snake segment in each space on a 28x13 gameboard. In principle developers should care about inefficiencies like this.

A valid counterargument would be that neither memory nor time is a concern for a game as small as this, and that the computer can handle it no matter what. This leads to the question of whether Pos makes the program more intuitive, expandable, modular, and so on. The answer is no, or not without being substantially changed somewhat beyond the course scope. In 100% of use cases, it is only important because of the data inside. Rather than simplifying existing interactions and making code easier to write, it only forces the unnecessary action of calling the x and y values inside.

[4 marks] If yes, discuss about an alternative objPos class design that you believe is relatively counterintuitive than the one in this project. If not, explain how you'd improve the object design. You are expected to facilitate the discussion with UML diagram(s).

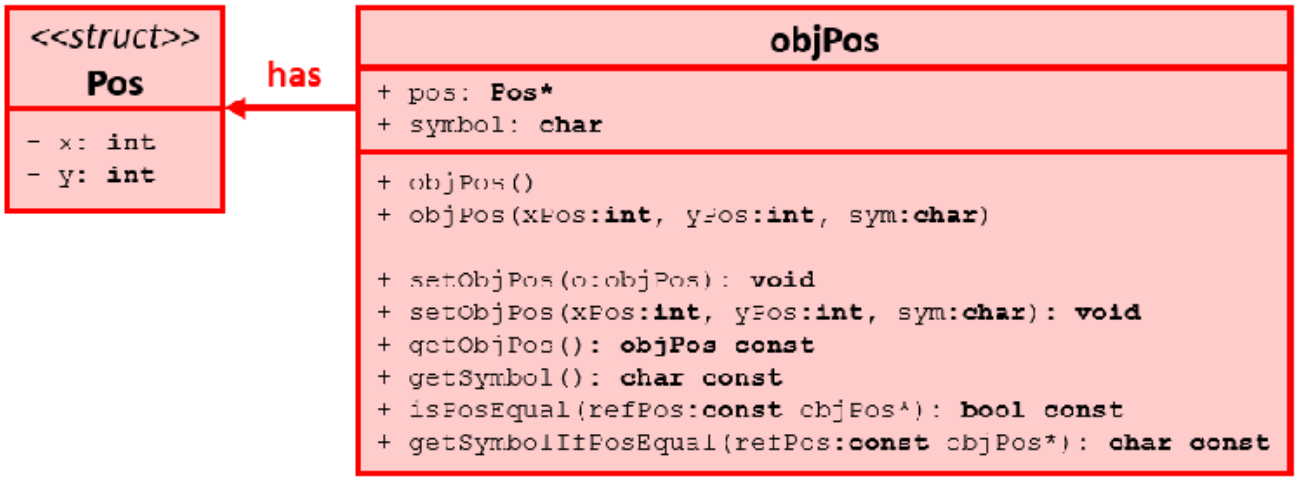

Figure 4. The implemented design, taken directly from the project description.

There are two ways to improve the design of this object. The naïve approach is to eliminate it entirely and instead define two new int fields in the objPos class, x and y. These could be public, like the symbol character in objPos already is. This approach would remove the overhead from the unnecessary member calls in the current implementation, while shortening many unnecessarily long lines of code. Additionally it bears mentioning that objPos does not necessarily need to store pointers to the coordinates because it

confers very little advantage for memory. The UML diagram below shows that this design is neater because it does not depend on interactions with a useless struct.

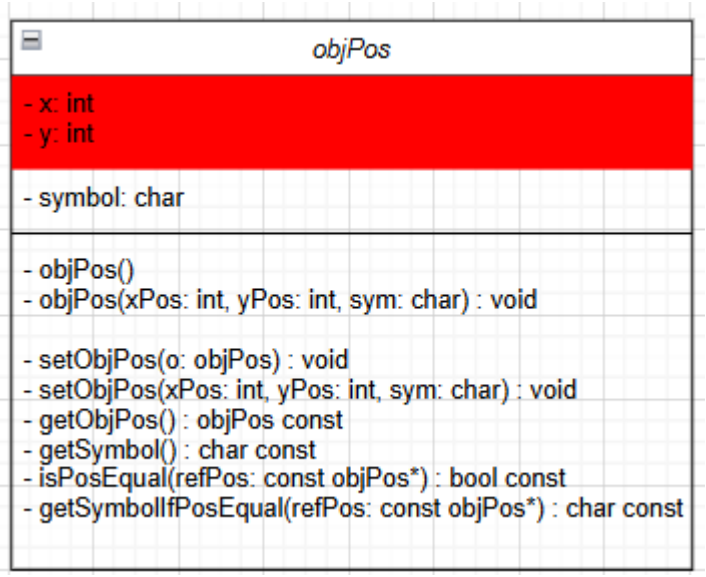

Figure 5. UML diagram of the first suggestion. Minimizes amount of interactions between objects.

The existence of Pos provides one possible advantage. It allows the positions of two items to be compared without explicitly referencing the underlying values, which would simplify a lot of code in this project. Overloading the equality (==) operator for Pos would provide a well-defined means of doing this, instead of forcing reliance on the limited ability of the compiler to compare structs. One could argue that overloading the operator for objPos would be more sensible, but it would need to disregard the symbol character to be meaningful, and that has the potential to cause confusion. Pos has the potential to unambiguously compare a set of coordinates, which would be quite useful in this project. This is what we would go with, because it is more readable than isEqualPos() and it gives purpose to something that was previously useless. The UML diagram below illustrates the ability to compare coordinates separately from all other data, given that the equality operator is packaged alone with the x and y values.

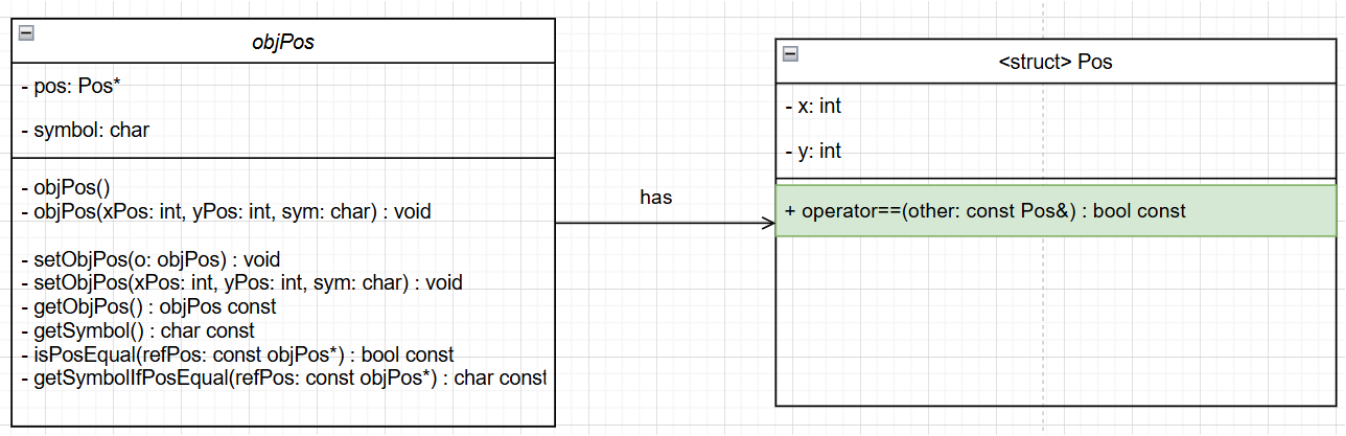

Figure 6. UML diagram of the final suggestion. Overloaded == operator allows a direct comparison of the coordinates of two objPos instances.