COMP ENG 2SH4 Final Project Peer Review

Our Team: Daniel Chirackal, Jaavin Mohanakumar
Team Evaluated: Azen Cardozo, Jay Tomar

Part I: ODD Quality

1.) The private qualities, and public methods within each class have clear names that allow for easy interpretation as to what each one does. When it comes to public methods like “generateFood ()” it is pretty self explanatory as to what it does. The input parameters of certain constructors are “pass by pointers” which reference other classes, this makes it easy to read how each class may or may not be interconnected. Whilst the code naming scheme is good the lack of commenting leads to some confusion, for example without knowledge or comments it might be hard to interpret what “getSymbolIfPosEqual(const objPos* refPos)” means without extra documentation.

2.) We can observe the main logic of the program fairly easily; the class to class interaction is fairly straightforward. All initialization of the allocated data is in “initialize”, also all importation of the class mechanics is within “run logic” and all “objPos” symbols/coords are in “Drawscreen”, so by inspection all steps of generation are in the respective places. When is comes to certain functions the mechanics exhibit little more procedural logic than “OOD” as shown in “Drawscreen” as symbols are sometimes explicitly stated. Furthermore there is a little bit of procedural logic in “RunLogic” that could be implemented into the classes, for example there is a special food that increases the players score by 10 explicitly states how its done however this could have been done with the class themselves.

3.)

Object Oriented Design		Procedural Logic	
Pros	Cons	Pros	Cons
The code functions, and main mechanics are compartmentalized to allow for easy team cooperation.	The importation of class mechanics into the main function and the other classes alike can be pretty troublesome as it can be hard to determine between pass by pointer/pass by reference.	All of the functions and main logic are set on one document that is readily available for manipulation. Making it somewhat easier to find semantic errors.	Since destructors cannot be made in procedural logic, keeping track of memory allocation/usage could prove to be more troublesome.
The inherent design of class allow for easy collaboration of mechanics in the main function.	Following engineering incremental design in OOD means the project may take longer than needed.	Since main mechanics like the FSM state machine, and player movement are already in the main code you don't have to worry about finding ways to import what would've been class functionality.	The inherent design of procedural logic does not allow for easy collaboration on projects, in fact it proves to be greatly difficult as there is no compartmentalizati on. This means that when uploading code to the repository there could be many conflicts between each partner's code.
Since the project is split up into different classes the work needed for each one is manageable.	Since the project exhibits classes, whenever there is an error (semantic) it can be hard to determine exactly what causes it.	The lower level design allows for more explicit instructions to be carried out. This means that the functionality of the code is buried less	Since the code for procedural programming is not separated by mechanics accordingly the main code can prove to be less

		in the class and more put on display so what the program can do is explicitly shown.	readable to foreigners.

Part II: Code Quality

1.) The code comes with a decent amount of self documentation, stating some of the intermediate steps of each function. However we would have preferred some more documentations on specific methods. In other words a small basic description of each method in all of the classes would've gone a long way in explaining the purpose of it relative to the rest of the class, and the project as a whole. This sort of description style could have also been used in the header files as well as the "Project.cpp". Despite this the functions mostly remain readable, and have their purpose in each step stated.

2.) The code mostly follows good use of good indentation of white spaces as variables, methods, and numbers alike are evenly spaced when it comes to mathematical operations. Continuing the code provides a good use of indentation of variable declaration, conditionals, loops, etc keeping indentations, and curly brackets where needed. There were only a handful of instances where this convention was broken, this happened in the main function, however this did not severely impact the legibility of the code.
Ex:

```
else if(myPlayer->checkFoodConsumption()==2) //if special food consumed
{
    //add 10 scores without increasing lengthen
    myfood->generateFood(tempPlayer);
    for(int i=0;i<10;i++) myGM->incrementScore();

}
```

Part III: Quick Functional Evaluation

1.) The game performs quite well and the game runs well. The food object generation is very good as the special food is well distributed across the board mixed in with the regular foods. The snake wrap around is implemented quite well. The player information like position and score is promptly displaced. Furthermore there are two games over screen which match the project requirements. However there was one small semantic error that was in the project.cpp file as pressing "t" in the program did not generate extra food. We believe this is the case because the command is referencing the game mechs class instead of the food class. So if they were to be referencing the food class where generate food is actually located the command would work.

2.)

```
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       3 unique,     3 total uninitialized access(es)
~~Dr.M~~       1 unique,    64 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total GDI usage error(s)
~~Dr.M~~       0 unique,     0 total handle leak(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
```

There are no memory leaks present in the final program. There however are instances of invalid heap arguments in the project.cpp file since there are "free" statements in the clean up function. Since there is a built destructor for the construction of an instance in all classes.

Part IV: Our Own Collaboration Experience

1.) The experience of working with my partner in this project was challenging but also very engaging. Despite the fact that updating, and commiting changes proved to be pretty difficult we had a fun time doing this project. The division of work in iterations, and the partner A and B structure made the project manageable.