Final Exam Readme
INFR 3110
Cody Jensen
100591285

Game Submission

Executable File can be found in the root folder of the github repository.
The controls for the game are listed below.

WASD or Arrow Keys to move the player.
R to reset the last 7 pellets back to their original state.

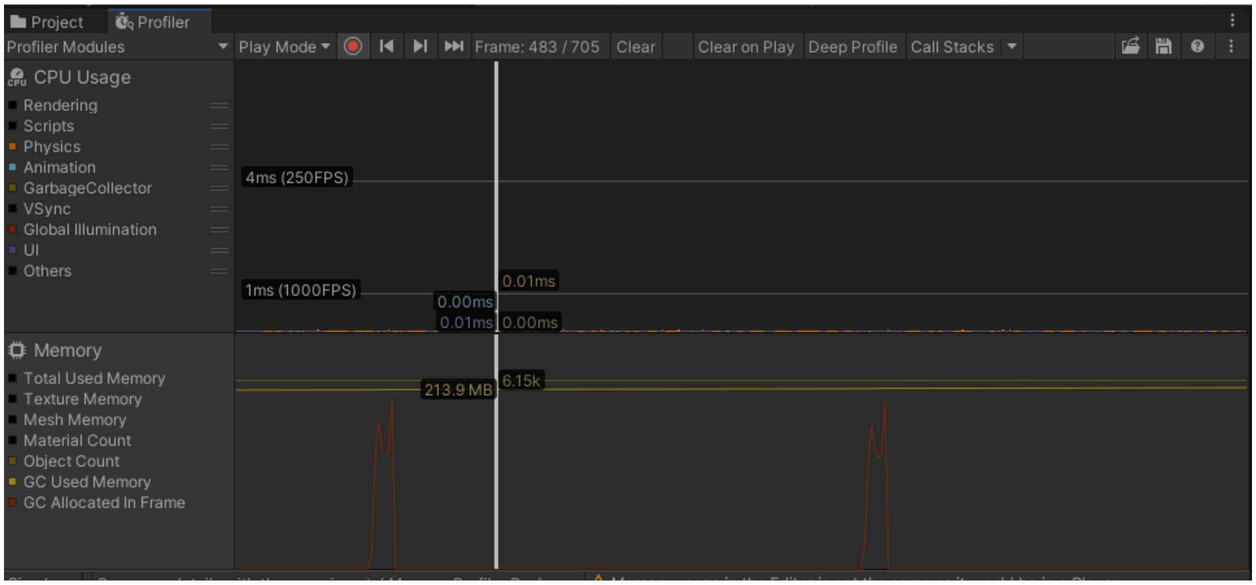
Win/Loss Conditions

To win the game, collect all the pellets. Upon collecting all 64 pellets in the scene, the game will transition to a win screen before prompting you to close the game.

If you for whatever reason make contact with a ghost, you will lose the game, resulting in a transition to a loss screen before prompting you to close the game.

Object Pooling

Object pooling was implemented to allow both the pellets and the ghosts to be generated at the start of the game as opposed to during play. It uses a single class to handle the pooling behavior, but multiple instances of the class are created to handle the pooling of different objects. For each list of objects I would like to have pooled, I create a new ObjectPool class and call the static Initialize function which allows me to determine what the object is and how many of them should be generated. I can then go into each individual pool and call the GetObjectFromPool function to return an object that is currently not being used. If all the objects in the pool are being used, I will instead get a return result of null, which should be factored into any attempt at accessing the pool to avoid errors.



In the screenshot above, I demonstrate the use of the object pooling for the pellets, but standard use of instantiation for the ghosts on a periodic timer. By looking at the GC memory allocation, we can see that the amount being allocated remains consistently low, with the exception of a few instances of large spikes occurring at different intervals. These spikes are represented by the instantiation of ghosts, resulting in an increased usage of memory allocation, which can place strain on the computer at runtime if too much is happening at once. Meanwhile, despite collecting pellets during this period of profiling, there was no indication of memory allocation for each pellet that was collected, as all of the pellets were allocated at the start of the game, as opposed to during the play experience. Without applying object pooling to the pellets, every time I collect a pellet, I would get another spike in GC memory allocation, which may affect the performance of the game.

Based on my profiling feedback, it is clear that Object Pooling is an effective management tool for a game like Pacman, as it can save a lot of memory and improve the overall performance of the game if we don't have to worry about instantiation so many objects, but instead can simply switch them on and off and reposition as needed.

Command Pattern

The command pattern was implemented into the game using three separate classes: the abstract Command class, the UndoCommand class which inherits from the Command class, and then finally the CommandInvoker class. The Command class is an abstract class with only a single function called "Invoke". The UndoCommand inherits from the Command class, and overrides the Invoke function with its own functionality. In this case, the UndoCommand holds a static list of GameObjects that is used to store the information of all the pellets collected by the player in order, so that the Invoke function can undo the last 7. Two for loops are used to achieve this result: the first is responsible for how many pellets can be undone (in the case that there are not a full 7 pellets) as well as which ones they are. The second loop re-enables these pellets and also removes them from the static list that tracks them.

This may be not only by the developer to test the systems of restoring the state of pellets, fruit, and enemies, but can also be permanently used by the game itself to restore the state of the game once a level is completed. The CommandInvoker does not have to rely on the player providing input for it to activate, but rather can use events to invoke a specific command, such as the previously mentioned completion of a level.

Explanation of DLL or Management System

Chosen Management System: **State System**

The state system is an important management tool that would be essential in the development of pacman. It is used in multiple situations within the game, including ghost state which determines whether or not the ghosts are hunting, being hunting, or deceased; the state of the player which determines whether or not they at a power pellet or if they are in default state; and finally the game manager itself which tracks whether the game is in a state of play or not.

To implement this, I would set up a few different states for the ghosts: hunting, fleeing, and respawning. The hunting state and the fleeing state would communicate to the AI controller whether or not the ghost should be running away from the player or chasing them. Meanwhile, the respawning state would have them remain in their cage while a timer counts down to return them back to their default state.

The player would have two states that it can toggle between, depending on whether or not they interact with a specific object within the game. The default state would render them vulnerable to ghosts, while the powered state would allow them to consume the ghosts.

Finally, the game state would track what state the game is currently in: playing, paused, menu, etc. This would be communicated to all objects and scripts that need to know this information, as it would not make much sense for ghosts to be able to hunt you down before you can even move.