Logan Miller

Final Project

Design

To start off this project I began by first designing the Game class and what objects it would need to function. In my initial design the Game class only had one player, an array of swans, a floor object and some integer numbers to save certain stats for the game. The functions for this class would mostly be for input validation, properly reading in/changing the maps, and maintaining the main game loop.

In order to get this game design to work I had to design a class for the floor and two actor classes for both the player and the swans. In the floor class I only really need functions that allow the 2d array of characters that make up the map to be read, print and set. It was around this point that I found it would make a lot of function calls easier if I defined a struct called location that would hold two integer values for the row and column of certain elements on the board. The actor class would only really need to hold the location of the actor and methods for moving and reading that location. By doing this, each of the classes that inherited the actor class would only really need to define their own move functions, along with any other supporting functions I may find necessary to implement along the way.

As it stands now I have a plan for the order in which I'm going to develop things. I picked this order as it seemed to tackle them in a way you could only achieve the next one on the list after the one before it is working properly.

1. Create and test the Floor class
2. Properly read in map.txt files and print them using the game class.
3. Create a player actor and move around validly.
4. Pick up and store apples and keys.
5. Open doors if there is a key in the inventory.
6. Make sure the swan tile moves the player back to start.
7. Apples can be moved.
8. Create an array of swan actors and move them.
9. Change levels.
10. Complete the game.

This order should allow me to linearly develop this program while insuring that all underlying process are working properly before adding logic on top of the assumption that they will work as they are designed to. This will lower the chance of huge logical errors running throughout my code due to fundamental levels working improperly. I plan on having three directories in my code. One for the headers, one for the maps and the last for the source code.

Implementation

The first issue I ran into in my development was getting the game class to properly size and read the map into its floor object. To get around the fact that my read function was not properly delineating the rows at the end of the lines I had the floor class first get each line into an array of strings and then write those strings character by character into the 2d character array. This allowed me to still be able to draw the maps and not have to put the whole map on the same line in the text file.

The next bigger design issue that I ran into after this was how I initially handled the player's movement and input validation. At first, since I decided to make and test the Play class first, I realized after getting the player class working that I had put too much of its movement validation process in the player class itself. This only became apparent once I tried to get the swans working correctly and realized that I had put the functions that didn't allow the player to move through walls and doors only in the player class. After some redesign of how the whole logic structure for the player class would work, I was able to put a lot of the bigger input validation functions in the game class as initially intended. The swans were then easily able to utilize the game class to make sure they weren't jumping through walls.

It was also during this whole input and movement validation process that I realized that my floor class definitely needed a copy constructor to deal with its 2d array. I had an inclination that this would be necessary at the beginning but had not initially coded it until I found that i needed to pass the floor around to a bunch of different functions to make sure that all of the actors were moving correctly.

The biggest thing that I encountered that I had not accounted for in my design however was a system to make sure that the player and swans left the character that they were covering up behind once they moved off of them. To account for this I added a new private member to the actor class that would hold the char that they were on top of. This would allow their move functions to be able to simply place down the character that was below them in that spot once they moved off of it. This solved that entire issue flawlessly and was easily overridden by manually changing the char below when the player actually picked up a key or apple. Assuming they had room in their inventory.

Beyond those issues is was just a matter of implementing some functions to insure that there were no actor characters where actors weren't actually present. This bug arose from some weird interaction that only happened sometimes when the swan and player would be occupying the same tile and then move off of it. Once that was solved though it was simply a matter of insuring that the maps would load correctly in sequence, implementing the code to bring in the number of levels from the command line, and making sure it printed a fancy victory message when you won!

Testing

Input	Output	Expected	Comments
Player attempts to move off the map from the starting position.	Invalid move. Step occurs.	yes	n/a
Player moves through valid open spaces on the map.	Player increments one spot at a time using W, A, S, D for direction.	yes	The starting E is properly maintained.
Player attempts to enter an invalid command.	Invalid move. No error message is displayed but	yes	n/a

	is prompted for a new command. Step does not occur.		
Player attempts to move into a “#”	Invalid move. Step does not occur.	yes	n/a
Player attempts to move into a “D” with 0 keys	Invalid move. Step does not occur.	yes	n/a
Player walks over an apple with 0 apples.	Apples increment. ‘A’ tile is set to a ‘ ‘. Valid move.	yes	n/a
Player walks over an apple with 2 apples.	Valid move. ‘A’ tile remains. Apples do no increment.	yes	n/a
Player walks over a key with 0 keys.	Keys increment. ‘K’ tile is removed. Valid move.	yes	n/a
Player walks over a key with 3 keys.	Valid move. ‘K’ tile remains. Keys do not increment.	yes	n/a
Player walks over a door with 3 keys.	Valid move. Keys decrement. ‘D’ tile is removed.	yes	n/a
Player walks into a tile that is adjacent to an ‘S’ tile.	Valid move. Player is teleported back to the ‘E’ tile. Map state and inventory persist through death.	yes	The decision to keep inventory on death was a design choice to avoid unsolvable levels.
The map contains an ‘S’ tile.	On any player input, the ‘S’ tile moves one unit in a random direction. If that move is into an invalid tile the ‘S’ tile remains where it was.	yes	n/a
The map contains multiple ‘S’ tiles.	All swans properly move on each step, and do not leave ghost swans behind due to actor interaction.	yes	This was the biggest cause of swan ghosts.

Player walks into an ‘E’ tile.	Invalid move. Step does not occur.	yes	n/a
Player walks over an ‘L’ or ‘X’ tile.	Valid move. ‘L’ or ‘X’ tile persists once player has moved.	yes	n/a
Player types in an E command.	Apples are decremented. Invincibility timer is set to 15 and properly decrements with each step.	yes	n/a
Player walks next to a swan while invincible.	Valid move. Player remains in this position and can move again from it.	yes	n/a
Player enters a U command while on an ‘L’ tile.	If total floors has been reached the game ends. Otherwise the next level is loaded, all swans are reloaded and all player stats are reset.	yes	n/a
Player enters a U command while on an ‘X’ tile.	The victor screen is shown and the game ends.	yes	n/a
Player enters a Q command.	Game exits with exit message.	yes	n/a
Multiple command line arguments are passed.	Error message is displayed and program exits.	yes	n/a
If a non-integer is passed as the first command line arguments	Error message is displayed and program exits.	yes	n/a