CS 162 Final Project Reflection

Hailey Wilder

Design Plan

In this game you are a young computer scientist lured to a haunted house with high hopes of scoring your first contract position. Things take a turn for the worst though, when you are trapped by your potential employer in his or her haunted house! You find yourself searching spooky rooms to discover the secrets of the house so you can escape. The goal is to navigate between rooms and obtain three items. One item you use to acquire an access code. Once you have acquired all the items and used one to obtain an access code, you can escape the house. There is a time limit since all that running about will tire you out, so navigate carefully and remember where you have been and what is there if you don't want to be trapped in the haunted house forever.

The player must navigate between rooms which are attached in a North, East, South, West configuration. There are no rooms connected at diagonals (i.e.: Northeast or Southwest) and a player can only navigate to an adjacent room. In other words, they cannot jump to a room that is not connected to their current room.

In order to achieve the goal of escaping the haunted house, the player must interact with the spaces. They must obtain a passcode and enter this code at a later time. They also must utilize objects they find in certain rooms in other locations. To further player interaction during the game, there is a time limit to encourage the player to be attentive to the game rather than just clicking about randomly. There should be plenty of time to complete all tasks though.

The main.cpp file is responsible for displaying the initial menu to the user and providing gameplay background. Thereafter, submenus, location updates, and hints are handled by individual Room classes. These also check to see if the player is staying within their time restriction.

When the Game class determines the player has won based on a status == ESCAPED, a message is printed by this class to let the user know they have won. This class also handles the case where the user runs out of time.

The Player class (which is assisted by the Game class) encapsulates current information regarding the player's in-game status. This class ties together inventory tracking, the player location, and timing.

Test Plan

The strategy for testing this program will be to implement the design incrementally, beginning with the simplest most independent functions and testing them as they are completed. The essential classes to begin with seem to be the Player, Game, Room, and Inventory classes. Once these are working independently, I can begin to build a simple main for basic testing. I also will use pre-conditions and invariants to guide my work. Therefore, establishing my file structure from the beginning will be very helpful and will allow me to work on smaller functions first regardless of which class they belong to.

Once basic functions are tested, I will build those functions that depend upon the tested functions only. These will be unit tested as coding progresses. Lastly, I will build functions that are interdependent upon multiple functions and/or classes. This will be the most complicated bit of programming because I will have to play the game to troubleshoot. In order to handle this, I will leave any features possible (any non-essential features for the game goal) until after testing. Once the more complicated functions are working in a satisfactory manner, I will begin implementing the rest of the features, providing myself with many print statements for troubleshooting feedback along the way. Eventually I will pair this down to just required game hints for the user.

I will test menu options by going through each of the menus and selecting every option iteratively to ensure there are not abnormal or unexpected events/ occurrences.

Finally, I will test game timing to ensure that the player has adequate time to complete the sequence of events required to win the game.

Test Results and Adjustments to Design

I generally was able to follow my design test plan and had few issues with unit testing the simple functions and those dependent upon them, although, I did need to wait to make my Game class until later than I thought I would since I needed almost all of my functions to be complete to know how to best construct this.

The Game class and the Player class ended up being the most difficult classes for me to implement which makes sense as they control most of the action. I resolved many of the issues with the player class by simplifying the code drastically. Initially, I had a lot of code written out in this file, but now it basically just calls functions from other areas. This works very well and keeps things running efficiently – I had a huge issue with memory leaks in that file to begin with. The biggest issue I had with the Game class was I had not taken the time to visualize or write out exactly what my inputs were going to be, so I had to revisit the design board for a bit to get that straightened out and then I was back on track. Again, here it was very helpful to call functions to keep my code concise.

Fortunately while testing my menus I had no issues. I did edit the names a bit to make sure that they were all consistent across Room subclasses. Looking back I probably could have had the menu written into the Room class somehow to save myself a bit of time on menu writing, but this method worked out too.

While I was running through my game, I found a few small issues with print statements and with how my haunted house was displaying, but those were trivial fixes. Unfortunately the lack of word wrapping on flip means there are word breaks during some of my back-story. I tried to prevent that as much as possible for the standard terminal size, but they are in there. The biggest thing I found during run throughs on my game was that the timing for end-game was way too long. My poor grader would have been sitting there forever. So, I shortened that up substantially – hope that is helpful!