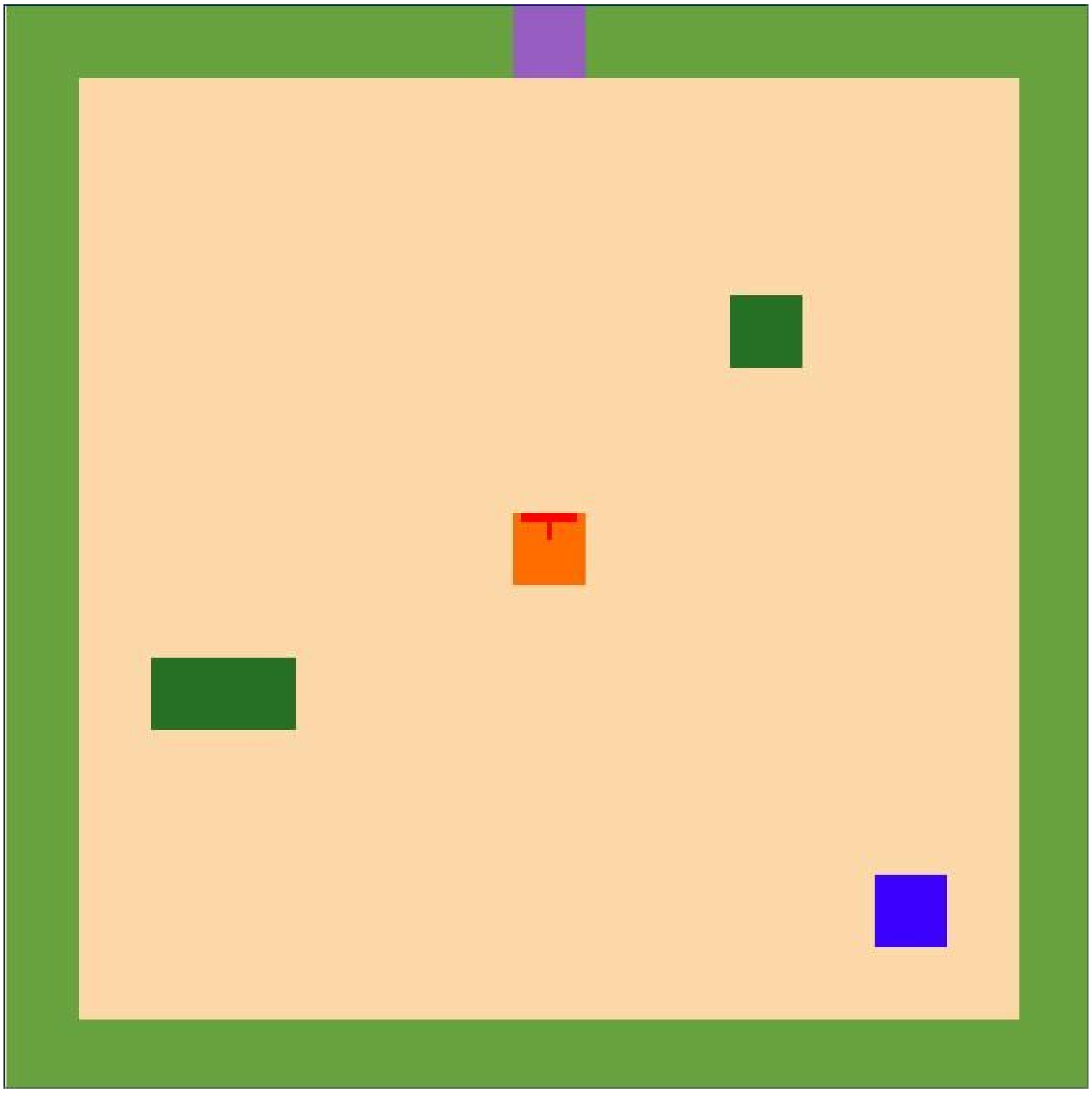
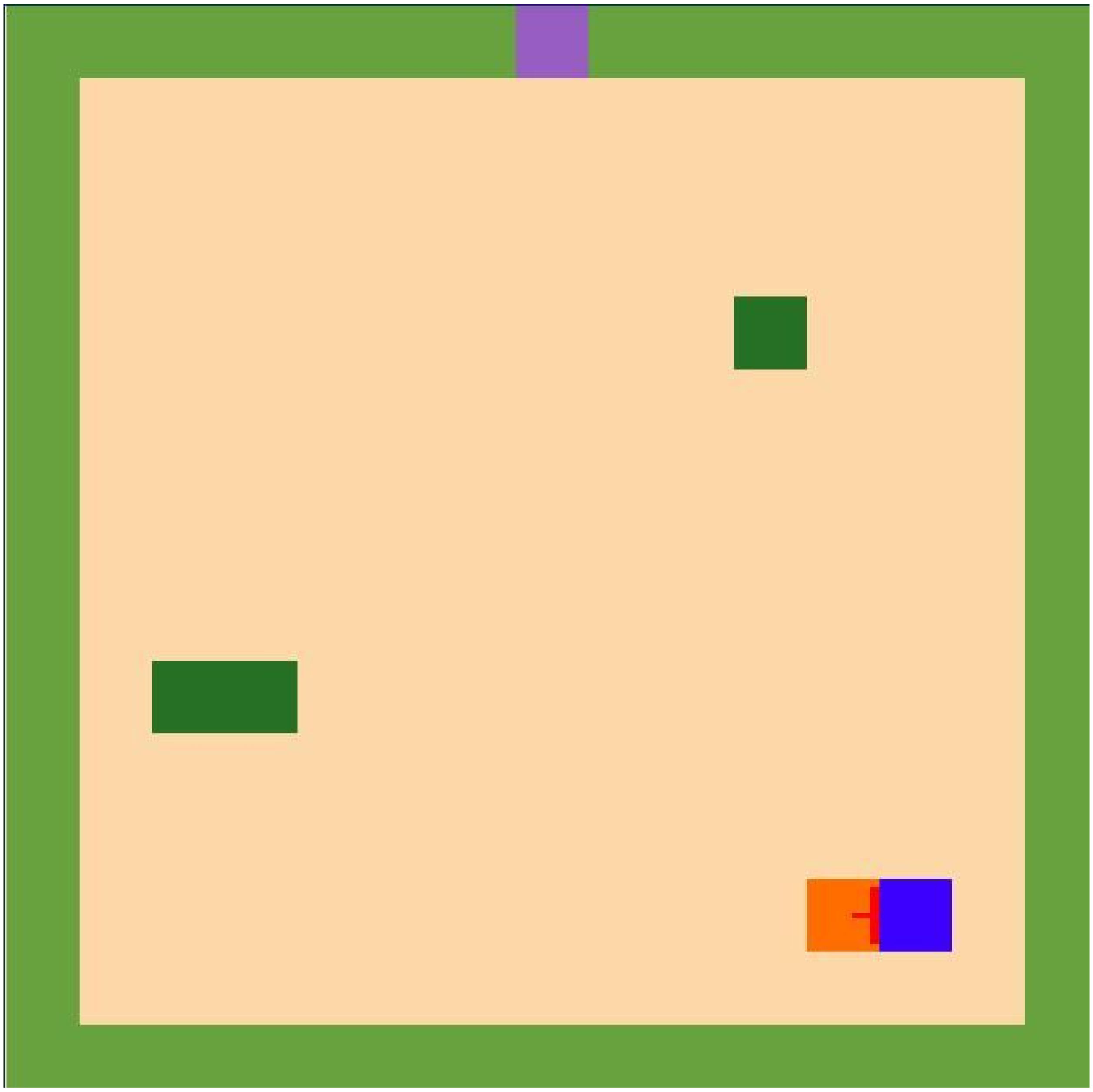
Project Overview

Our project is a small roguelike RPG where the player controllers a character using the keyboard to defeat enemies in the room before moving onto the next room. The door, obstacle, and enemy locations in each room are randomly generated.

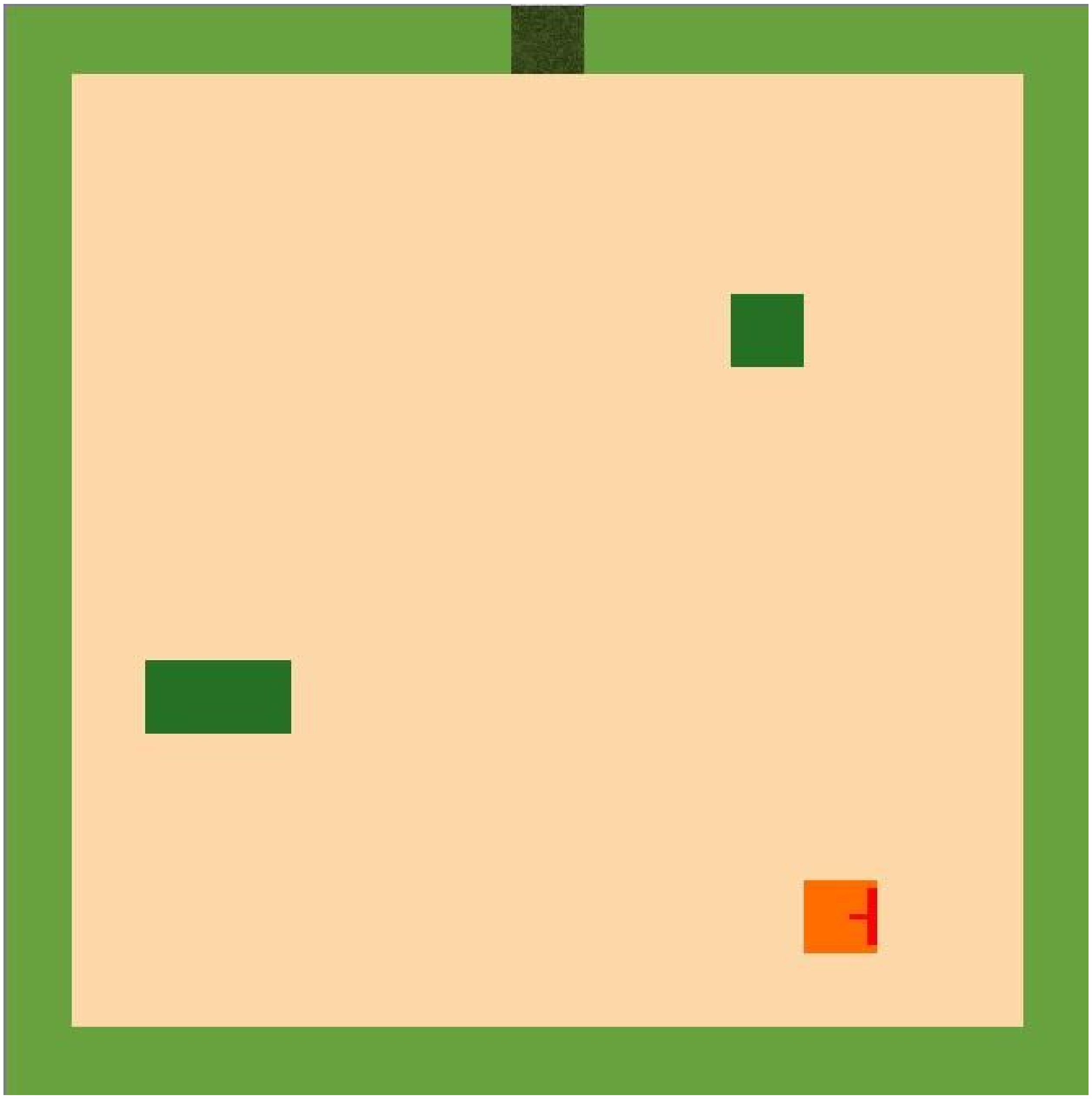
Results



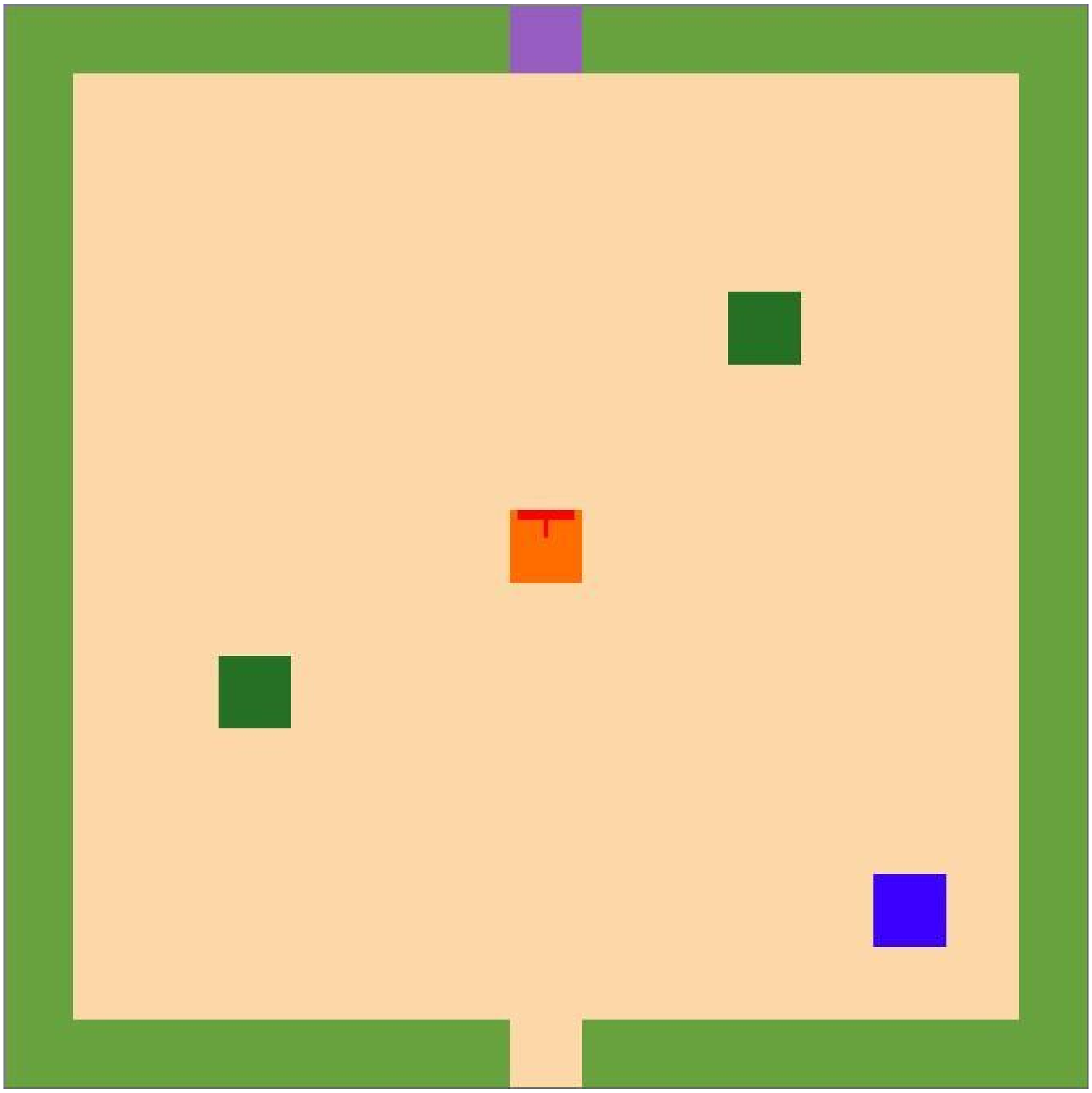
Our program creates a random world with a random collection of hills and an npc in one of the corners.

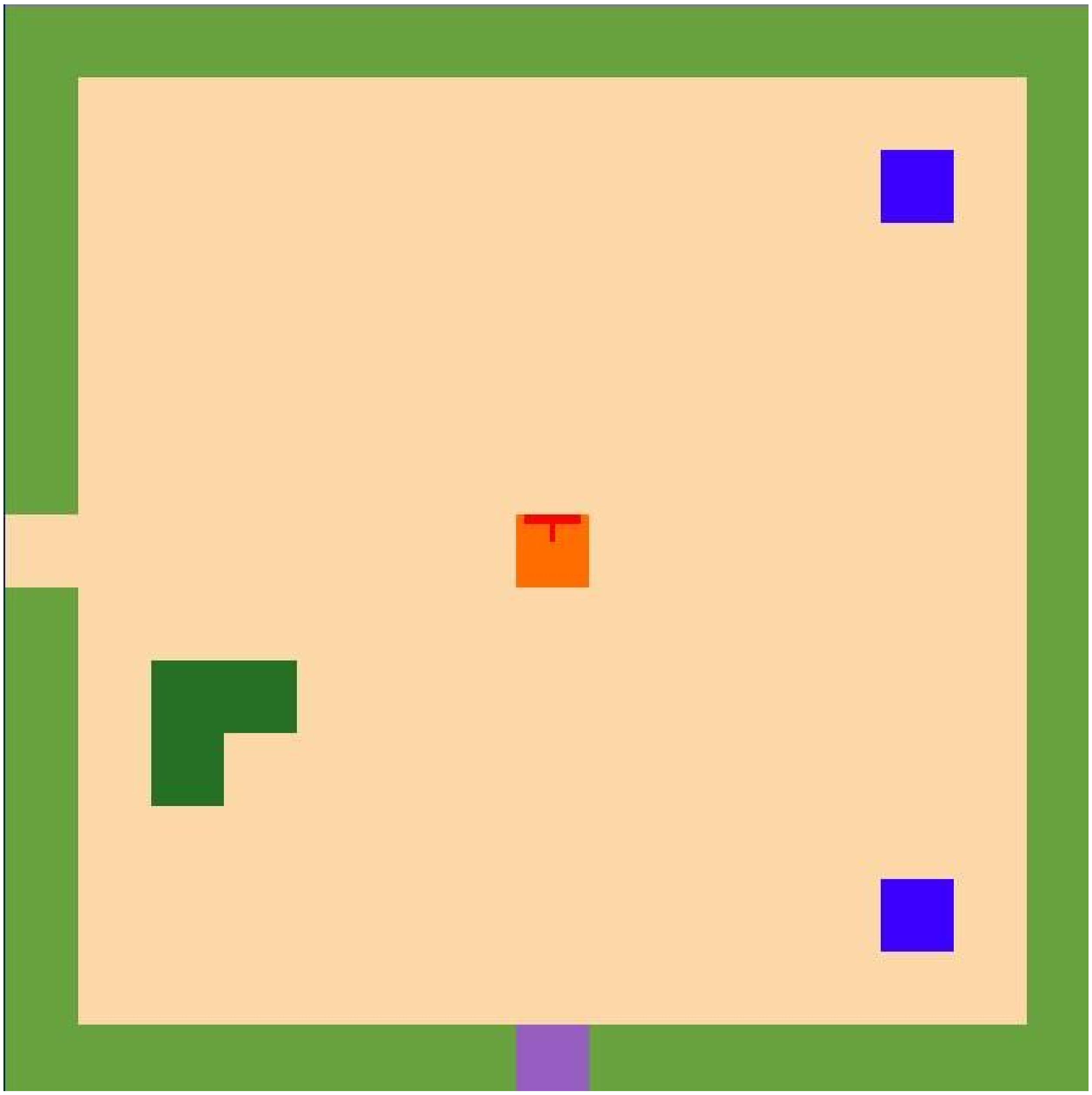


The player can move in the program using the arrow keys, WASD, or ,AOE (Dvorak WASD location). The player can also rotate using these keys (pressing rotates and holding moves).



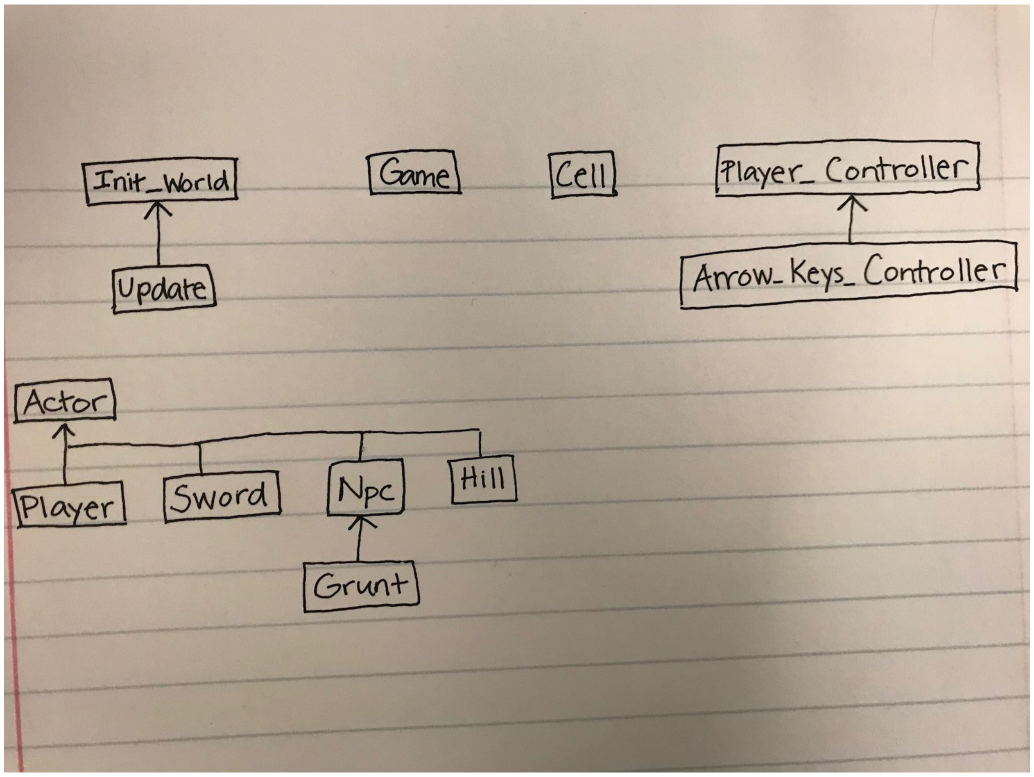
Using space, the player can attack with a sword to hurt the NPC. Once the NPC reaches 0 hit points, it dies. Once all of the npcs are killed the door opens (see the top of the screen).


The player can go through the door to enter the next room.


Starting with the third room, the number of NPCs increases to two NPCs.

Implementation

For our implementation we decided to have four separate modules. The main module, game.py, defined the Game class and ran the game, calling the other three modules to do so. The module controller.py defines the Player_Controller class and the Arrow_Keys_Controller class which inherits from it. These two classes handle the player input from the arrow keys and the spacebar. The module gameworld.py deals with creating and updating the world of the game, and contains the classes Init_World and Update, which inherits from Init_World. The final module, actors.py, defines all the actors to be placed in the world and the cells (the Cell class) which construct the coordinates of the world. The classes for all the actors are defined by a main class, Actor, and more specifically by the classes Player, Sword, Npc, and Hill. The class Grunt inherits from Npc because it is a specific kind of Npc with additional functionality.



One of the biggest design decisions we made was in the creation of the Game class. Originally, all of our code ran out of the if __name__ == "__main__": part of the code. However, one of our stretch goals was for our game to have multiple rooms. Our original way of approaching this involved nested while loops (two running loops) and using the World module to manage everything that was positional (IE the side of the door). This was unorganized and resulted in inappropriate and irrelevant functions ending up in the Init_World class within the world module (items that did not have to do with a single world). This also meant it was hard to keep track of what the current level was, which made changing the npcs based on levels difficult.

By creating the Game class, we were able to keep track of the current level, the opening position, and the door position all outside of the world method classes. This made more organizational sense as these items have more to do with the game, and not the individual world. The Game class also provided organization for checking events during the game, which allowed the run world function in the game module to be shorter and easier to read.

Reflection

Overall, our project went well. It was appropriately in scope, and we got the majority of our work toward producing the MVP over spring break, so we were able to spend a good amount of time debugging. We learned a lot about game architecture and how to develop a game, and in the future would be able to make better decisions about designing a game. It would have been helpful to know more about the broad architecture of how to develop this kind of game, because some of our decisions about how to implement movement and our use of the cell grid were more complicated than they could have been and were too integrated into the code by the time we realized that there were simpler ways to implement them.

We didn’t formally divide the work, and often took turns working on implementing parts of the code and debugging that we could see solutions for. We also implemented pair programming using Teletype.

In regards to unit testing, we did not use any doctests due to the difficulty of using them to test in our program. Our functions were tested during implementation by using print statements to print the values that the function was modifying or creating. We also were able to recognize when there were issues with our functions when issues showed up in our game window.