Group O: Dylan Brown, Jackson Hoyt, Cole Rife, Alex Wisser

Dungeon Dweller: Program Description

The code for Dungeon Dweller contains three explicit classes. The Player class keeps track of the player's stats and location on the game board. The Monster class allows the developer to easily create monsters with various levels; the class will take in a number which represents the level of the monster, and it assigns the monster's stats based on the level. The last class, RoomChecker, allows the developer to keep track of the room throughout all stages of the game. These are handle classes so that the variables assigned to those classes can be edited outside of the class itself.

Variables:

Some of the variables are assigned in order to assign values to the sprites. These are the door_sprite, blank_sprite, boots_sprite, player_sprite, sword_sprite, potion_sprite, shield_sprite, monster_sprite, and empty_room. These values are based on the location within the retro_pack.png file. The gameboard_display is a vector that keeps track of the game board and which sprites should be placed in which spaces. The spriteVector variable is a vector containing all of the sprites. This is created for easy iteration through all of the sprites. The my_scene variable is used in order to draw the game board figure.

The playerObj, monsterObj, and room variables are objects that are used to track the player, monsters, and floor number respectively. The monsterObj is declared each time a battle occurs, while the playerObj and room variables are declared once at the beginning of the code.

As opposed to recreating the playerObj and room variables, the values assigned to these objects are changed when necessary. This is the benefit of making these handle classes.

The variables within the loop are the running, result, isChanged, oldXPos, oldYPos, keyInput variables. The oldXPos and oldYPos variables are used to track the position that the player was located before moving. isChanged is used to track whether or not the player has moved. The running variable is used to track whether or not the game should continue after the end of the loop. Result is used to track the result of a battle if a battle occurs. The keyInput tracks the input by the user from the keyboard.

Within the Player class, the variables health, level, xp, attack, defense, speed, xPos, yPos track the attributes of the player object. The monster class only contains health, attack, defense, and speed; these are also used to keep track of the monster's stats. The room class only contains the room variable which tracks the number of the player’s current room.

Aspects of Code:

The code begins with the initialization of all the variables. It assigns values to all the sprites and sets the gameboard_display to a blank room so that new sprites can be added later. The code then prints instructions on how to play the game alongside a little bit of backstory to the game. After this, the code begins the creation of the first room. The first two steps of the room creation are to place the door in the bottom right corner and place the player at the position (1,1). The code then loops through the sprite vector and generates two random numbers that

represent. The numbers have been restricted to 2 through 9 so that the sprite cannot overtake the player or the door sprite. After the loop concludes, the new scene is drawn.

Within the main while loop, which runs while the running variable is set to true, the oldXPos, oldYPos, and isChanged variables are initialized. The keyInput is also initialized with the value of the keyboard input. The value of the keyInput is checked to see if the player has hit a button that corresponds to movement. If the button pressed corresponds to movement, the program checks whether or not the movement is allowed. If it is allowed, the player's position is updated to reflect that change. After the movement occurs, the program checks whether or not the player has moved onto a space containing a power-up. If the user has moved onto a space containing a power-up, the player's stats are adjusted accordingly. For example, a sword power up would increase the player's attack stat. If the player has not moved onto a power-up, the program checks if the player has moved onto a space containing a monster. The player then moves onto said space and the board is redrawn.

Battle:

If the player has walked on a space that contains a monster. The program enters a battle. To begin the battle, the program prompts the user to select attack, defense, or speed. After the player's input has been stored, the monster's input is generated by a random number generator. The winner of the battle is then determined. Attack beats defense, defense beats speed, and speed beats attack. After the battle is determined damage is calculated. The minimum damage is 2. Damage is calculated by 5+(winning stat - losing stat). If the damage is less than 2, the damage is reset to be 2. Damage is then applied to the loser and the new health is displayed. If the

monster's health goes below 0 the monster is removed from the board and xp is awarded to the player. The xp gain is based on the health of the monster. If the floor number is less than the player's level then the xp is the maximum health of the monster divided by 2. If not the xp is simply the maximum health of the monster. The program then checks if the player has leveled up. If the player has leveled up, which occurs when the player has obtained 20 or more xp, the player's xp is reset to 0, the player's level is increased by one, and the player's stats are all increased by 5. If the player's health goes below 0, the game is over, and the player has lost.

If the player reaches a door, a new room is generated. See first paragraph of **Aspects of Code** for the steps of room generation. If the new room would be the 6th room, the player has to battle the boss. The boss battle operates the same as a normal monster battle; however, the boss has a starting level of 8. Before the boss battle the program loads in both an image and an audio file. The program then displays the image and plays the music. To see the steps of battle see **Battle**. If the player wins the boss battle, the player has won the game; however, if the player loses the boss battle the game is over. After the conclusion of the game, the windows are closed and the result of the game is displayed. The music from the boss battle is then stopped.

Game Commands:

One of the game commands used is the draw scene method from the SimpleGameEngine. This creates a GUI game board based on the gameboard_display vector.

The other game command that is used is the getKeyInput method. This method loops while a key is not pressed. When a key is finally pressed, the method gets the source of the input and returns it.