ASCII GAME

Clayton Sweeten

April 28, 2015

1 Introduction

The intent of my project was to create a basic, terminal based, game. These games use the terminal to display basic graphics, and interpret user inputs from the keyboard. All of which is done dynamically. In 1980 a game with this idea was created, Rogue, and from then on all games of this style are known as Rogue-Likes. These games are characterized by using ASCII characters as graphics, having multiple areas or dungeons, having a basic battle system, and items. These games require input from the keyboard, and then update the terminal based on the input. That type of dynamic control of the terminal window is not possible in basic C++. However, a library has been created to do just that, namely, ncurses. The implementation of this game relied heavily on the proper use of ncurses. Ncurses is actually quite intuitive; however to a beginner it can seem quite complicated. To alleviate this complexity, an online tutorial resource was used.

2 Curses Basics

Curses is a library used to manipulate the terminal, and interpret key commands. It does this by storing a type of structure called a WINDOW. These windows represent the terminal screen using a coordinate system of rows and columns, with 0,0 at the top left of the window. To add characters, strings, numbers, etc. to the terminal screen, the user specifies a new location for the curser, an invisible entity that points to a specific place on the screen. Ncurses then places whatever is specified in the argument at that location. Because of that, most ncurses functions take in arguments in the form: what window to print on, where to print, and what to print. Knowing these it is easy to use most ncurses functions. Some notable ones are, mvwaddch() which moves the window and adds a character to the specified place,

mvwprintw(), prints to a specific window, and wattron() which sets certain attributes to characters, like bolding and color. Most importantly, whenever any of these functions are called, the window must be refreshed, using wrefresh(). Otherwise nothing gets printed to the screen. There are two functions that are primarily used for keyboard input, getch() and mvinch(). Getch() stores whatever key press it detects in a character. And mvinch() moves the window and returns the character it found there. These two functions where used heavily in the creation of the game.

3 Notable Difficulties

3.1 Sub Window

One main idea of the game was to have a small sub window that scrolled when the player moved. That way the entire game map wouldnt be displayed at once, which could not fit on a normal sized terminal screen. Once this was done, using the function subwin() the task was to create a function that moved the small window with the character, effectively keeping the player centered on the middle of the screen. The idea behind this is simple, if the user presses left, move the character to the left and then move the window to the left. However, difficulty arose when the player got to the edge of the map. The sub window would continue past the map, and break entirely. The following procedure was implemented to fix this problem. When the character moves, check if they can see the edge of the map. If they can then move the window back by however much distance they have moved (call it delta). This way, every time a player moves towards the edge of the map, delta increases, and therefore the window is moved further back, keeping it from going passed the edges of the map. This was implemented in the function center().

3.2 Enemy Movement

An enemy that stands still and waits for the player to attack it is not very interesting. So a function was needed to move them towards the player. This was accomplished by a series of if statements, to test first if the player was within some radius of the enemy, then check which side of the enemy the player is on. The first part was done using the basic distance formula between two points:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1}$$

formula here The second part was done by subtracting the distance of the enemy and the player, and checking if that value was negative or positive, which told the enemy which side the player was on. The problem with this function arose when the enemy was moved within the function. For some reason this simply could not work. So instead the function returned a value, based on the calculations performed, then outside the function the enemy was moved with another function. This made it so there were two lines of code necessary for each enemy to move. This doesn't sound like much at first, but when moving multiple enemies it became quite cumbersome. Which lead to the overwhelming lines of code that are present in the game loop() function.

3.3 Battle

An enemy that cant attack the player also makes for a rather bland game. This function was created so a player and enemy can interact. The chosen method was a dice roll. Both entities rolled a dice, whoever got the higher number won. This was rather easy to implement. However, since the movement function is called many times in the game loop, even after an enemy was defeated they would be recreated once the movement function was called on them. To solve this problem, in the battle function, once an enemy was defeated their new position was defined to be some place far outside the map. This is a rather silly way to accomplish this, and led to further disorganization in the game loop function. Not only that, but to discover which enemy needs to be moved, if statements were placed inside the game loop alongside calls to the battle function; for each enemy. This complicated the game loop a moderate amount.

References

[1] Pradeep Padala, *NCURSES Programming HOWTO*, tldp.org/HOWTO/NCURSES-Programming-HOWTO/

[2] *Rogue(video game)*, en.wikipedia.org/wiki/Rogue28videogame29