Connor Fitzpatrick

Alexandria Sisk

Eduardo Vazquez

CSE 130-50 Introduction to Programming Languages

Final Project – Group 9

Introduction

The project described in this paper is that of a text-based linear RPG (role-playing game) designed to be an introduction into the field of game design. There are simple end steps in the game that stop your progress at certain intervals of pathing and specialized combat/encounters that give the game a feel for story and decisive battles. The project began by the division of labor so that no one person had too much work on their hands. Connor Fitzpatrick headed the task of making the foundation of the game, menus, and combat. Alexandria Sisk headed the creation of paths and how they interacted with encounters. Eduardo Vazquez headed the creation of the weapons and their stats.

Developer Files

Main.cpp

This file contains the main method for the program. It begins by seeding the random generators with time(NULL) so each time the rand method gets called, it generates a truly random number. By utilizing the Menu class system, the initialization of the Main menu object occurs, and its main method gets called pulling the program into the Menus.h file. Once the player has opted to start a new game, the program exits the main method for the Main menu object and calls the newGame method to continue execution.

Menus.h

The core thing to note in this file is throughout the game the menus allow the player to select a specific action. These menus are a series of classes that all share the same attributes and methods. The parent class is the Menu class where four attributes are declared and defined, which can be seen Figure 1. menuScreen is a Boolean attribute, it executes in each menus main method that determines how long the menu will be present on the screen. sel is what is known as the select bit or the current selection. This integer holds the current selected option the cursor is on to print the proper position of the cursor on the menus. maxSel holds the maximum number of selections/options are available per menu. These integers are counted starting at zero, i.e. If there are four options, then maxSel is three and sel can be anywhere from zero to three. input holds the player's input from their keyboard to call methods from the Controls.h/cpp files. handler holds the action it receives from the listener methods, all designated in the Controls files. Two virtual methods are created for inheritance to other menus. printMenu is the method used to output the

specific menu to the screen. menuEvent is the method that determines what action needs to be taken for the program to continue depending on the menu's options. Every screen has the same menuMain or main method. Inside menuMain is a while loop that iterates through until menuScreen is false. It will clear the screen, output the menu, obtains the input from the getKeyboardInput method, figures out the position of the cursor and puts it in sel, finds if the user selected an option and holds the choice in handler, finally, it calls the menuEvent method.

```
// Menu class is the base class for all menus in this game
class Menu {
public:
    // Defines several attributes, each with unique properties:
    // menuScreen - True if the menu should continue to be shown, false otherwise
    // sel - The current selected menu option in the menu
    // maxSel - The maximum selection the menu holds
    // input - Holds the keyboard input from the player
    // handler - Determines what actions should be performed next
    bool menuScreen = true;
    int sel = 0;
    int maxSel = 0;
    string input = "";
    string handler = "00";

    // Default constructor used mainly for inheritance purposes
    Menu() {}

    // Virtual function to be overloaded in each inherited class,
    // printMenu does as its name suggests, prints the menu that the player
    // is currently on
    virtual void printMenu() {}

    // Virtual function to be overloaded in each inherited class,
    // menuEvent determines what to do with the handler attribute
    virtual void menuEvent() {}

    // menuMain is the main method for every menu class
    void menuMain() {

        // While menuScreen is true
        while (menuScreen) {

            // Clear the screen
            system("cls");

            // Print the designated menu
            printMenu();

            // Get the keyboard input
            input = getKeyboardInput();

            // Determine where the cursor is for the sel attribute
            sel = arrowHandler(input, sel, maxSel);

            // Determine if the player has chosen an option for the handler attribute
            handler = enterHandler(input, sel, maxSel);

            // Call the menuEvent method
            menuEvent();

        }
    }
};
```

Figure 1: Screenshot of the Menu class

From this parent Menu class there are the following child classes:

- Movement
    - Combat which inherits Movement
    - Selection which inherits Movement
    - Prologue which inherits Movement
    - Final which inherits Movement
- EndTesting
- Main

The reason four additional grandchildren classes inherit from Movement is, so the initialization of a Player pointer does not have to occur in each class. From here on, the above classes will be referred to as menus.

The Movement menu establishes the connection between the menus and the players. It also controls the execution of movement during the game. Its menu is pictured in Figure 2. The player gets the option to move left, forward, and right. There are precautions in place inherently in the Player class which will be explained further in this report. Movement also controls whether the player has lost all their life and outputs a "Game Over" screen before exiting the game.

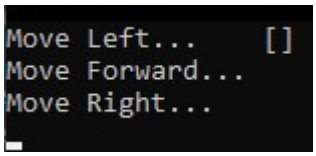

Figure 2: Screenshot of the Movement Menu

Weapons, prior to Combat

The team created a weapons class that held three attributes they are accuracy, damage, and range unfortunately the team did not have enough time to implement the range attribute as this would have taken extra time to program and add more complexity. These attributes are standard with any weapon that will be listed, each weapon is either accurate or not, and deals a great amount of damage or not. The weapons class will be the parent class for the weapons the player will use for the game. The following weapons are as followed: Axe, Sword, Bow, Knife, Bat, Gun, and Shield. Each weapon is a class itself that inherits the weapons class thus inheriting the attributes as well. For each weapon that is listed, there are different attributes within that class, for example, in the game if you select Axe you can select to throw or swing. If you select Bow, then you can either load your arrows and shoot or hit the opponent with just the Bow. If you select Gun, you can throw the gun at the monster or shoot at them. The different attributes listed under each individual weapon gives the game an actual gameplay on what action to take against the opponent.

Inside main() is where the game starts calling our classes so that the player has the option to choose and select their weapon. The player starts by choosing a weapon from the list that is provided to the player. Depending on the weapon choice the program will go through a few ifs, else if, and else statements. As an example, if the player chose Bat the program will go to the if statement *else if (weapon == "Bat")* and start printing the information inside the if statement.

Menus.h continuation

The Combat menu controls the combat in the game. The specific mechanics of combat are in the Entities.h file. However, this menu is extremely important as it holds different actions

depending on what the player's weapon is. In this menu, the player will generally have two actions, attack or run. Attack will deal damage to the opponent but then the opponent gets a chance to counterattack. Run allows you a 40% chance to escape the fight without harm.

The Selection menu is primarily for the player to choose their weapon for combat throughout the game. There is also more to the Selection menu as it implements the option to test combat for debugging purposes.

The Prologue menu is to show the player their starting stats.

The Final menu is designed to ask the player if they are ready to finish their journey or to continue. This menu is also in charge of healing the player every time they reach a potential ending.

The EndTesting menu is a menu to test the four endings of the game.

The Main menu is the first menu that appears when you launch the game. The player gets the option to start a new game, test the endings, test combat, or exit the game entirely as pictured in Figure 3.

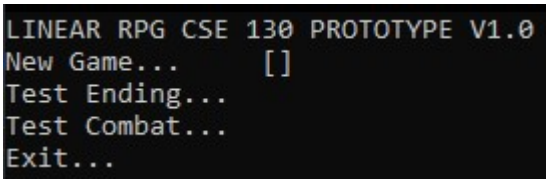

Figure 3: Screenshot of the Main menu

Controls.h, Creation.h, and Ending_X.h are only used to initialize their respective methods.

Controls.cpp

The importance of this file is to activate the listeners of the game. Specifically, getKeyboardInput, it pauses the program until a valid keyboard input has been received and determines what specific key was pressed and returns it to wherever it was called. The exception being the "escape" key that exits the game at any point. arrrowHandler has parameters of an input string, the sel option, and the last or maxSel option. Then, depending on where the cursor is based on sel, changes its position based on the input string. enterHandler takes the same parameters and outputs the handler in terms of a string as "XX" where the first 'X' is one or zero, one being activated, zero being not, and the second 'X' being the sel option.

Creation.cpp

This file contains three methods: combat, towardEnd, and newGame. Combat is a method that is specifically created to circumvent the modulization of menus and the order of inclusion files. I receive C2065 errors without doing this. towardEnd takes parameters of a Player pointer, Movement pointer, and a path integer. This method iterates until the player has moved the max number of paths and outputs the Final menu to end or continue. It outputs if the player has decided to end the program. newGame instantiates a new Player object, Prologue, and Movement objects with that player address. Then a for loop iterates through until the player decides to end or does the max number of paths.

Ending_X.cpp

These files are specific to each ending that run each ending. They hold specific outputs and actions dedicated to each ending 1-4.

Endings.h

This file has a class hierarchal system for each ending. It has the main End class which gets inherited by the four subsequent classes and executes their main methods to execute the endings.

Pathing.h

I used randomization for every time the player moves to simulate at 50% chance nothing happens, a 20% chance that the player experience an encounter, and a 30% that the player enters combat. Within these are more randomizations like the level of the slime for the combat, the type of encounter, etc.

Pathing continues

Regarding our pathing for the project, we decided to go with using classes for our encounters and our endings. The first class created was the encounter class. The encounter class includes methods to access data members. The first method is the Encounter construct, which uses a construct to allow for delivery of the player's name via pointers and the trail variable. The construct passes said variables to our second method, void callTrail(). Void callTrail() goes into the if statements such as *if (trail == 1)* print what is inside the statement. In our project, each of the if statements check to see if they were called from our randomized encounter function. If

they are called it will print out the story they were assigned. The endpath class follows a similar set up as the encounter class. We created the endpath construct in the class to deliver the variable ending to the callEnding function. The callEnding function then cycles through if statements, for example, *if (ending == 1)* print the corresponding ending, for this example ending one will print. This repeats for each of the four endings in our code.

Entities.h

This file contains all the types of entities in the game and the attack mechanic. The attack mechanic, specifically the attack method, has parameters of a template class A (attacker), template class V (victim), and a style of combat. Inside the method there are specific actions for specific weapons and whether the weapon hits based off their accuracy and the randomly rolled hit counter. It then outputs a pointer to the V class so that the game can directly manipulate the class without instantiating new ones. Then the Player class and Slime class enter where they have constructors to create a player with 500 health and a Slime with a parameter of level to determine its health using an algorithm of

$$hp = level * (rand() * 199 + 1) \quad (1)$$

Conclusion

While the game may not be as polished and as amazing as initially visualized, the game far exceeds the team's expectations of their abilities and the possibilities of creating such a game. Overall, the team is happy with the outcome of the game. For example, there was supposed to be more variety in monsters, however, due to a miscommunication, the teammate assigned to the creation of entities was not able to complete their task.

Throughout this experience, the team has learned a lot. It was everyone's first experience in creating a game, let alone a multi-thousand-line project in C++. The team is immensely grateful for the opportunity and will take our experience and teachings from our professor, Dr. Pamela Thomas, to heart and continue furthering our knowledge in the days to come.