Lecturer:

Jude Joseph Lamug Martinez, MCS

Made By:

Ariel Prandi Darmawidjaja - 2702337584

BINUS UNIVERSITY INTERNATIONAL

JAKARTA

2023

TABLE OF CONTENT

PROJECT SPECIFICATION

My project is an attempt to recreate Pokemon with JavaSwing. The goal of the game is to defeat as many wild pokemons as you can before your health depletes and you run out of pokemons.

The game is run in a main game loop (while loop) that calls the update and draw methods, which will respectively update information regarding game elements and draw game elements onto the screen. The game will only have Charmander and Bulbasaur as the two available playable pokemons; the only available wild pokemon is Bulbasaur. The game has two different overall states: In Battle and Not in Battle.

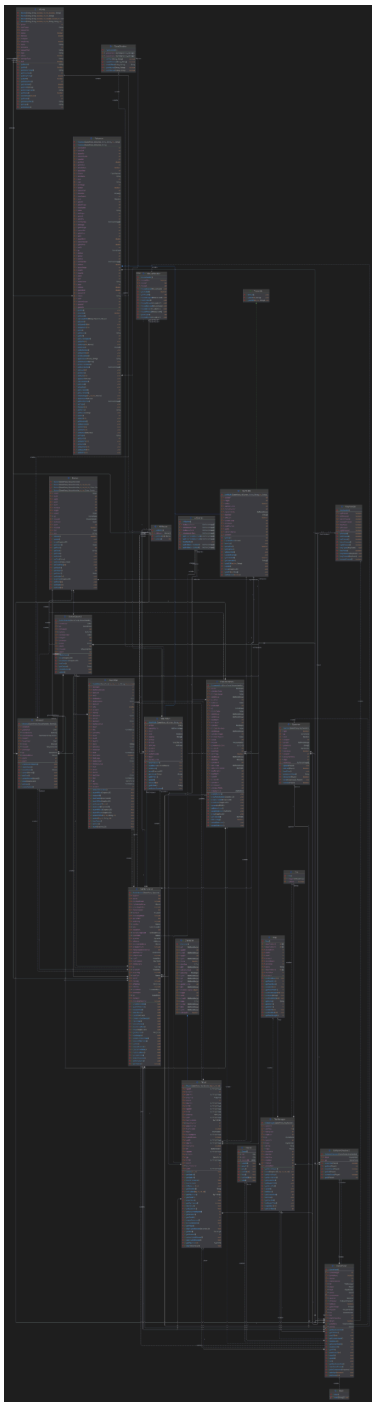
In the overworld state (In Battle), that is when you can freely control the player character and walk into the tall grass by using the classic 'WASD' control scheme; players can view their pokemons by pressing the 'P' key. In the tall grass is where the player has a random chance (one in nine) in encountering a Bulbasaur. If a Bulbasaur is encountered, the game state is swapped from Not in Battle to In Battle.

In this new state, the player information in the overworld stops updating and being drawn as well as the map. On the other hand, the battle screen starts to be updated and drawn by the BattleHandler class. The BattleHandler class is where the player's pokemon and the wild Bulbasaur can battle and duke it out to the death. The player can select whether to fight, switch pokemon, or run from battle by interacting with UI elements by clicking onto them with the mouse cursor. If the wild pokemon has fainted, it breaks out of the In Battle state and returns you back to the overworld state. However, if it is your pokemon that faints, you will likewise return to the overworld state but the pokemon that fainted is removed from the inventory. If the

player has run out of pokemons to use, they cannot encounter wild pokemons and the background music will slowly turn demonic.

SOLUTION DESIGN

i. Class Diagram



ii. Features implemented

a. Main Game Loop

The main game loop is in the GamePanel class where all information regarding the game is updated and drawn. The frame rate is set to 60FPS by using nanotime; time of the first loop is recorded and then the thread is paused by Thread.sleep for the remaining time until the next update cycle. The main game loop also helps manage states of the game, it will only update and draw the player and the map when not in battle, and update and draw the battle screen when in battle.

b. Map Generation

The generation of the map is mostly handled by three classes: Tile, Map, and TileManager.

The Tile class is a short class, only consisting of two attributes: image, and collision. The image attribute is a BufferedImage data type and it will store the image of the tile. The collision attribute is a boolean and it stores information of the tile's status as an object that can be collided or interacted with.

The Map class has essential information regarding the map such as the number of columns and rows (in terms of tiles) and the starting position of the player. However, most importantly the map class has three 2D arrays which store the tile number in the 2D array respective to their column and row in the CSV file.. The tile number is the number on each cell of the CSV file, where the map is read, which corresponds to the index of a tile in the array which contains all tile images. There are 3 tile number 2D arrays: tall grass, ledges, and miscellaneous.

Each tile of the map is drawn relative to the position of the player in the world; the player is most of the time drawn fixed at the center of the screen.

```
while (worldCol < this.maxCol && worldRow < this.maxRow) {

    int tileNum = this.maps[0].mapTileNum1[worldRow][worldCol];
    int tileNum2 = this.maps[0].mapTileNum2[worldRow][worldCol];
    int tileNum3 = this.maps[0].mapTileNum3[worldRow][worldCol];

    int worldX = worldCol * gp.getTileSize();
    int worldY = worldRow * gp.getTileSize();
    int screenX = worldX - pWorldX + pScreenX;
    int screenY = worldY - pWorldY + pScreenY;

    g2.drawImage(allTiles[tileNum].image, screenX, screenY, gp.getTileSize(), gp.getTileSize(), null);
    if (tileNum2 != -1) {
        g2.drawImage(allTiles[tileNum2].image, screenX, screenY, gp.getTileSize(), gp.getTileSize(), null);
    }
    if (tileNum3 != -1) {
        g2.drawImage(allTiles[tileNum3].image, screenX, screenY, gp.getTileSize(), gp.getTileSize(), null);
    }

    worldCol++;
    if (worldCol == this.maxCol) {
        worldCol = 0;
        worldRow++;
    }
}
```

Code snippet of tile drawing

As seen from the snippet of code above, the screen position where the tile is drawn is the difference between the world position of the player and tile added with the screen position of the player in order to see the distance between the player on screen and the where the tile would be on screen.

c. Player Movement

The player character is able to move in the overworld by using the ‘WASD’ control scheme. First and foremost, the game is able to take keyboard input by adding KeyListener to the JPanel by adding the class KeyHandlerwhich implements the KeyListener interface.

The player has a set movement speed, that is approximately 4 tiles per second. When moving, the player’s position will remain constant at the center of the screen most of the time. The movement is mostly done by moving the tiles surrounding the player relative to the player’s position in the map.

The movement is tile-based, meaning the player's position will always be situated within a tile and not in between tiles. The way this is done is by taking advantage of the fact that my game has a set draw and update rate of 60 frames per second

```
if (this.moveCounter == 17) {
    this.moveCounter = 0;
    this.moveDone = false;
    if (this.worldX%gp.getTileSize() > 0) {
        this.worldX = this.worldX - 1;
        }
    this.staticX = this.worldX;
    this.staticY = this.worldY;
}
```

Code Snippet from character movement

There are two attributes of the player class which helps in achieving that snappy tile based movement. First of all, by trial and error, I discovered that it takes approximately 17 frames of key press to move one tile. The moveDone boolean becomes true when the player is stationary. When it is false, it indicates that the player is still in motion. Essentially, the game will only accept keyboard input every 17 frames to emulate snappy movement.

d. Collision

The collision is handled by a collision handler class. The collision handler class checks for the tile that is one in front of the direction where the player moves; it checks for the collision boolean mentioned beforehand in the Map Generation section. If collision is true, the tileChecker

object will return false and the player cannot move in that particular direction. Otherwise, they can still move freely.

e. Pokemon Encounter

Pokemon encounters are mostly handled by the Spawner class and helped by the Collision Handler class (to detect whether in grass or not). The spawner class uses the collision handler class to detect when the player enters a grass patch. If the player enters a grass patch, a random integer from 1 to 9 (exclusive) and that number represents the number of steps (recorded from when the player first enters grass) it takes for the player to encounter a pokemon. If the step count (in the grass) is equal to the randomly generated integer, then a pokemon is encountered. When a pokemon is encountered, the game will switch states into the battle state, where player info and map info is no longer updated and drawn.

f. Game States

The game has two different states that are represented by the boolean attribute in the GamePanel class called inFight. inFight is false when the player is exploring the overworld and not in an encounter. When an encounter occurs, inFight becomes true and the main game loop will stop updating and drawing the player and map. The game states also help in playing music. Different music will play depending on the game state.

g. Sounds

Sounds are made possible by the Sound class which consists of the Clip attribute and URL array attribute. The URL array stores the information on the music or sound effects as input stream in an array; the Clip attribute helps in converting the input stream into an audio file that can be played when the game is run. The music choices I was able to implement due to being able play sound were the light hearted and adventurous overworld music and the upbeat and hype battle

music. The only sound effect I used was the thud that played in the actual pokemon games when the player collides with an object that can be collided with.

h. Pokemons

Pokemons are partitioned into two different types (classes): MyPKMN and WildPKMN, which are classes for pokemons that belong to the player and wild respectively. Those classes inherit from the Pokemon super class.

The Pokemon super class contains many attributes that mostly just represent the stats of the pokemon, its movepool, its name, and in-battle stats and stat modifiers. The Pokemon super class stores information that differentiates the different pokemons from one another. For example, the base stats of the child class of Pokemon are determined by a method which contains a switch case which determines the base stats of that class by getting the name of the pokemon. All sprite information is also stored in the Pokemon super class.

There are key differences between the MyPKMN class and WildPKMN class. First and foremost, MyPKMN objects are able to gain exp and level, getting stronger in the process. On the other hand, WildPKMN objects don't have that but instead have methods that will randomize its stats to differentiate each encounter from another. Wild pokemons will learn moves from an excel file depending on the name of that wild pokemon. It will learn moves from its current level to the 4 moves below its level, and if the level isn’t high enough to learn 4 moves the blank moves will be left null and unusable in battle.

When the player pokemon’s health reaches 0, they will be forever unusable, and when the player runs out of pokemons to use they can never enter battle again.

i. Battle

Battle is handled in the BattleHandler class. This class acts as a stage for the player's pokemon and wild pokemon to be placed in. The class has two attributes just for this: myPokemon - representing the player's currently selected pokemon, and wildPokemon - the current wild pokemon being faced. The myPokemon and wildPokemon attributes are initially set as null. The reason for this is the spawner method.

The spawner method checks to see if myPokemon is null. If it is, then it will assign my selected pokemon to the attribute and send him into battle. The spawner method also checks to see if the wildPokemon attribute is null. If it is, then it will spawn in a wild pokemon.

The BattleHandler class made use of many booleans to create separate states within the battle. The attributes are: choosingAction, choosingAttack, and choosingPokemon, animationTurn. In addition to those attributes, I had also created some boolean flags for the actions that the player might perform: running, attacking, and switching.

ChoosingAction applies when the player is choosing whether the controlled pokemon should 'Fight', 'Run', or switch to a different pokemon. If one of the actions is selected, the ChoosingAction boolean is set to false and the action that is chosen has their respective flags set to true.

The animationTurn state is a unique flag that essentially makes it so some animations and sound effects are possible in battle. It helps indicate that there is a speed hierarchy. If my pokemon has a greater speed than the enemy pokemon, then the wild pokemon is damaged first indicated by the flicker animation. Otherwise, it is my pokemon that is damaged first, indicated the same way. It also allows for the health bars to update after the animations play, instead of immediately after choosing an action, to try and stay true to the original games.

The player can only exit the battle if either one of their pokemon dies or the wild pokemon dies. The BattleHandler class will check for the health of each of the pokemons every turn and promptly return the state into the overworld state and exit battle.

j. Buttons and UI

The Buttons class made use of the MouseListener interface and the MouseMotionListener interface to get information regarding mouse position and click status. I designed the button class in such a way that I can easily customize its positions and dimensions very easily and also set its colors for when it is not hovered over by the mouse cursor and when it is hovered over by the mouse cursor. In addition, it can also return a boolean to give an indication if it is being clicked which was very useful when designing the menus.

The UI that made use of the button class included: The action chooser menu when choosing what to do in battle, the move choosing menu when choosing what move the pokemon would use, and the pokemon switch screen. All of which made use of the Button class and I made them correspond to their respective options depending on the UI. For example, I made each button correspond to the pokemon's move in the choosing move menu.

The last UI element was the health bar of the player's pokemon and the wild pokemon. It didn't make use of the button class but only made use of the Rectangle class to draw the health bar.

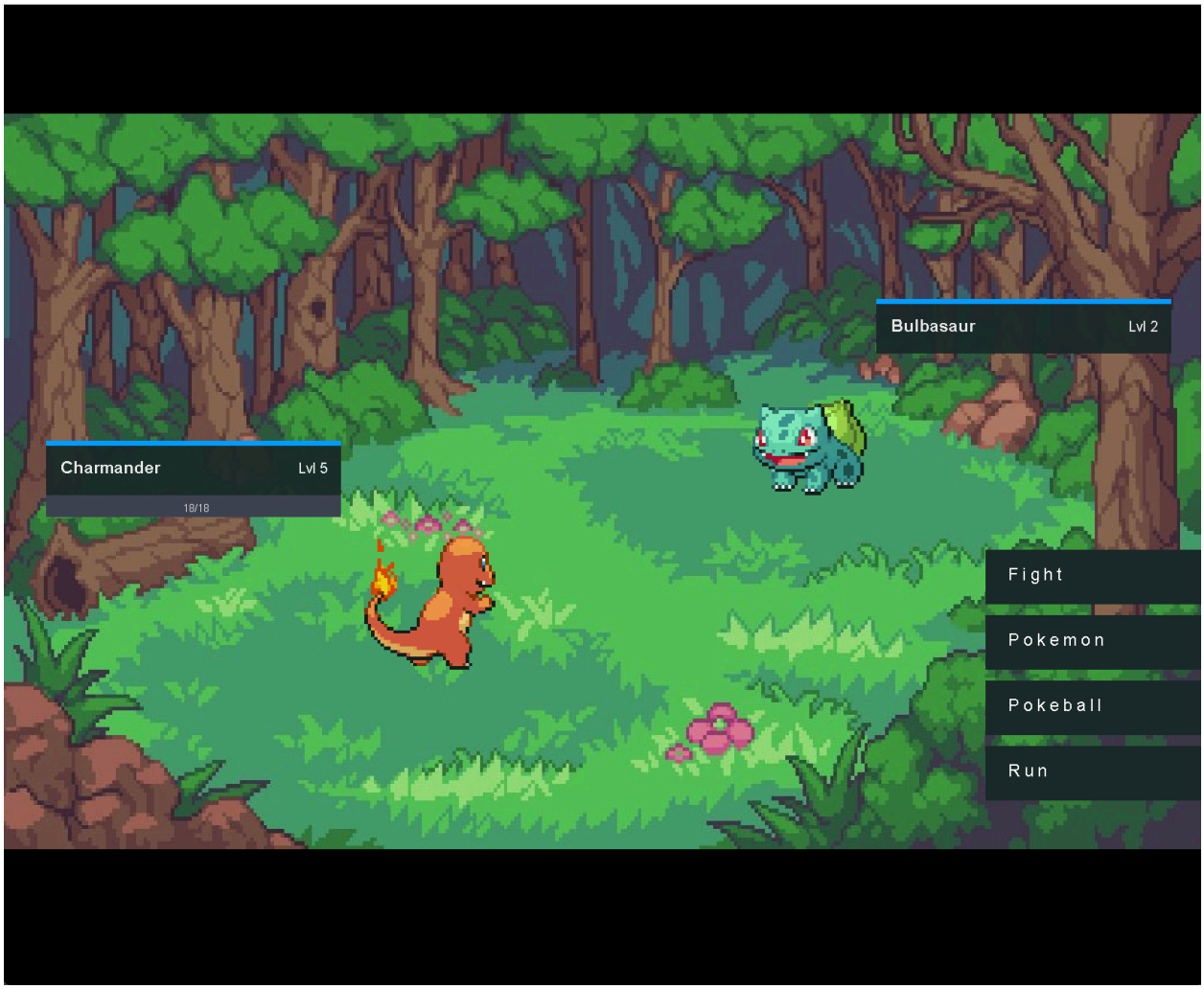
k. Data Structures

The only data structures I made use of were Arrays and HashMaps. Arrays were used for nearly everything; some examples would be the pokemons the player owns, the pokemon moves, and the sprites for each pokemon. The HashMap was used only for problems regarding pokemon typing. I would use a HashMap to help determine which type was strong against what type.

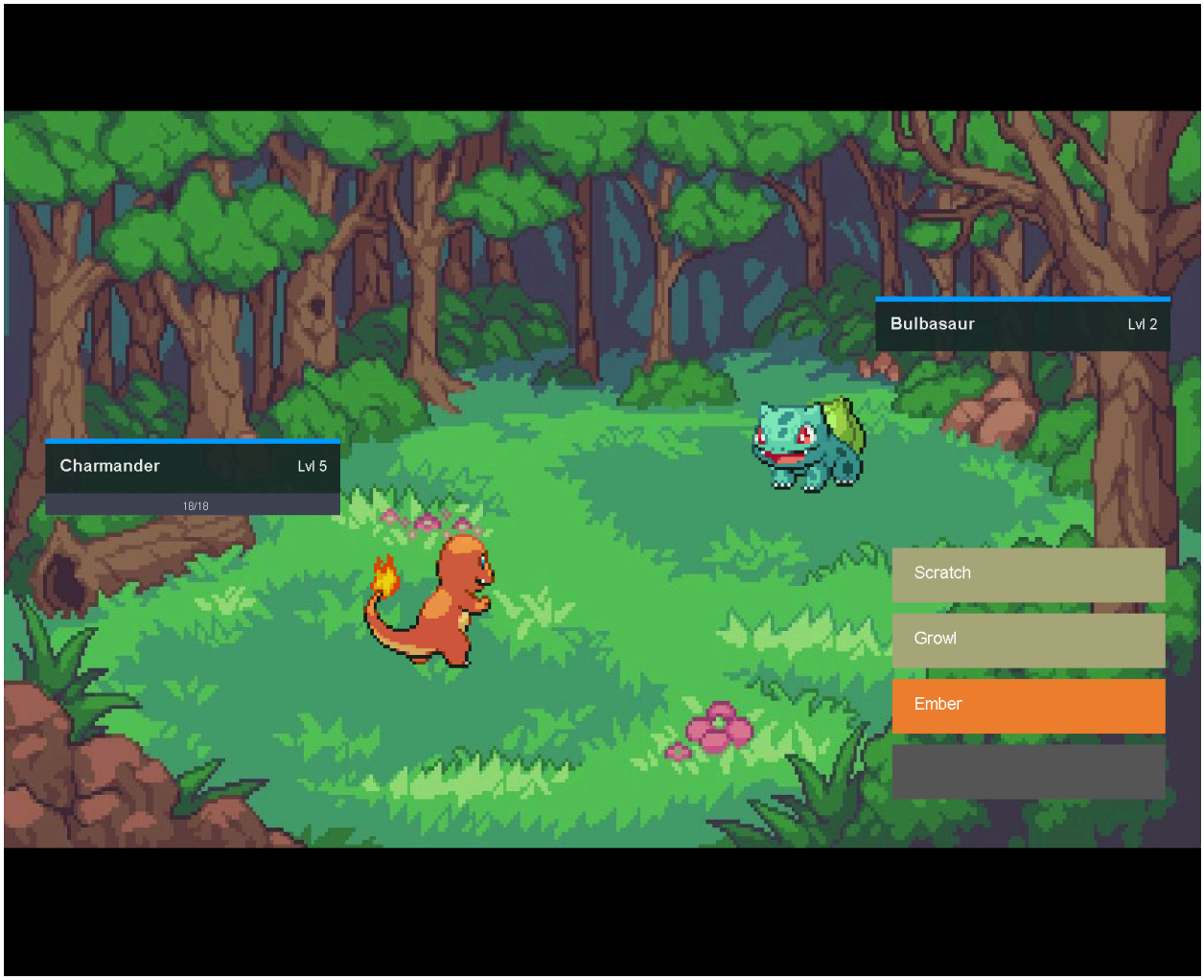
iii. Evidence of Working Program

The start of the game

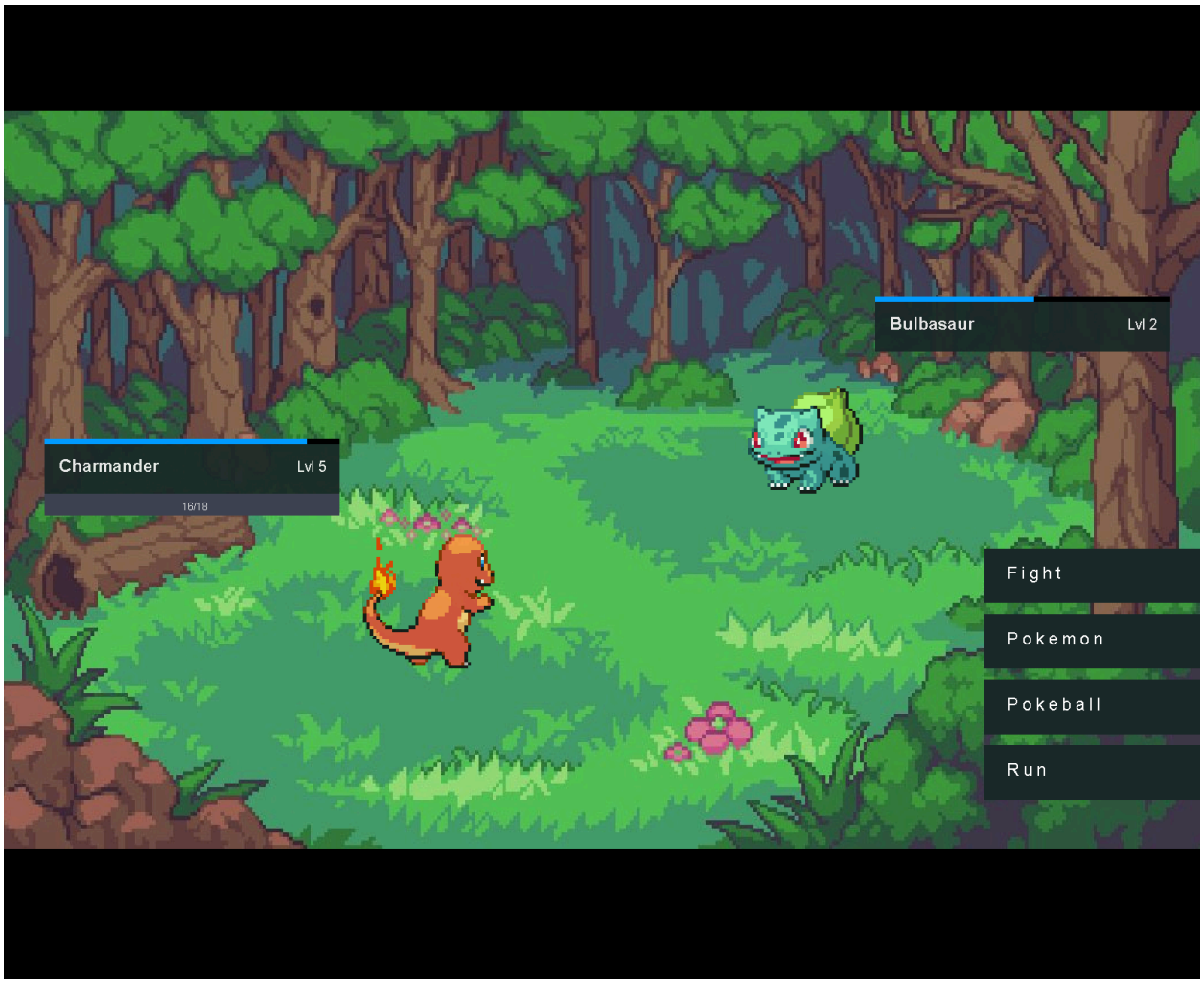
Movement and collision



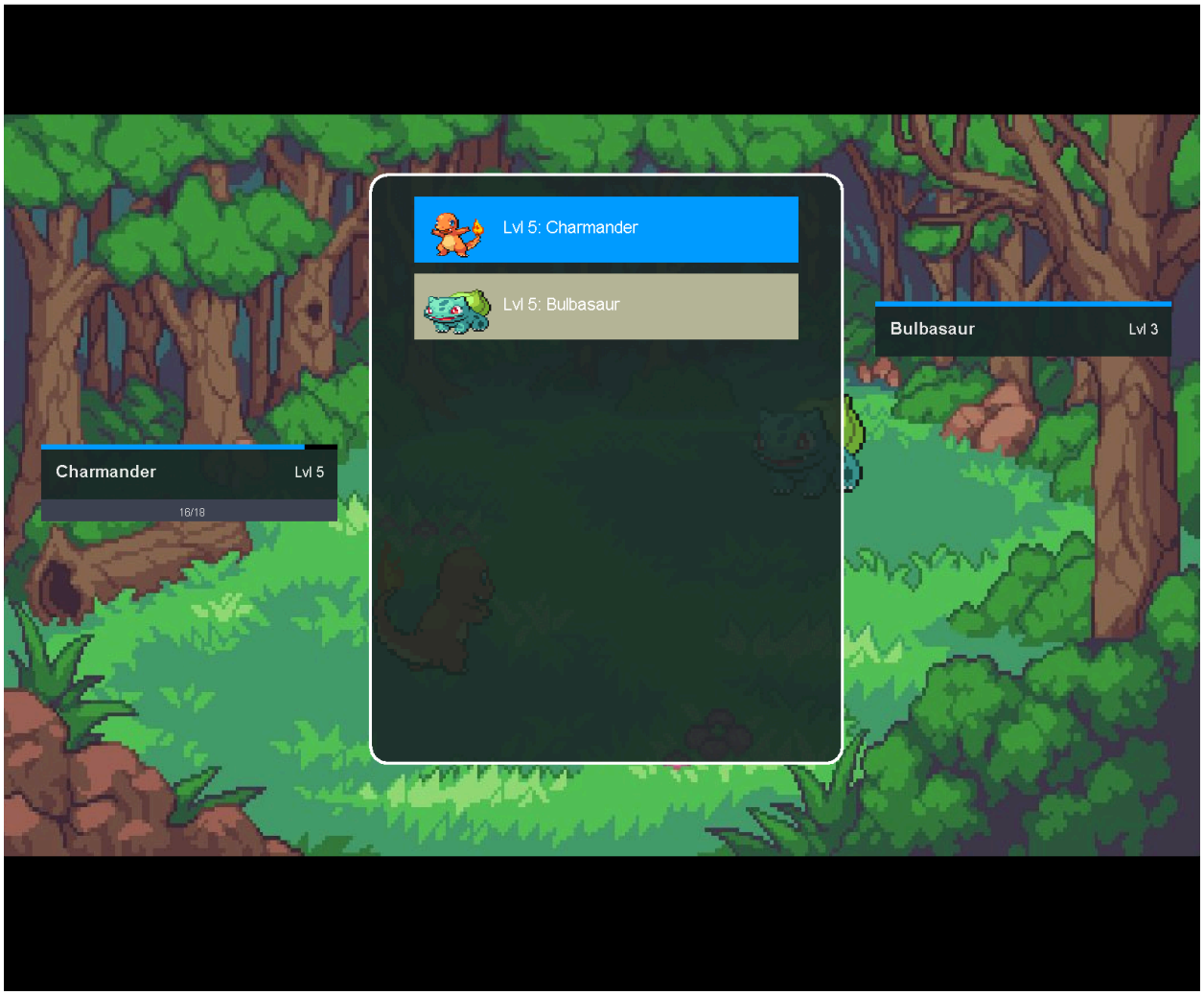
Encounter and choose action UI



Choosing a move to attack with, the choose moves UI



Pokemons taking damage



Choosing to switch pokemons

iv. Resources

Game Making Tutorial (for the basics of javaSwing and MapTileGeneration, inspiration for player movement and tile collision):

https://www.youtube.com/@RyiSnow/videos

Game Assets:

https://www.spriters-resource.com/game_boy_advance/pokemonfireredleafgreen/

https://www.spriters-resource.com/ds_dsi/pokemonblackwhite/