PROJECT PLAN: POKEMON SIMULATOR

Alexander Kernot, Hoang Bao Chau Nguyen, Roselyn Hoffmann

University of Adelaide 2023

Contents

1. Use Case of simulation

This project aims to be a recreation of a generation 1 Pokémon battle simulator akin to Pokémon showdown for entertainment-purposes. It will be a turn-based player versus computer game where players either construct a team of 6 Pokémon, or they will be randomly assigned Pokémon according to constraints that will depend on the game mode played. The game modes that will be implemented are as follows:

- Standard mode; 6 Pokémon with moves that are valid in Pokémon Red will either be chosen by the player or assigned randomly, following basic tournament rules.
- Random battle mayhem; the player will receive 6 random Pokémon that are given 4 random moves, ignoring any constraints, and will have their type modified to match the typing of the Pokémon's first two moves. Furthermore, the type matchups will be reversed. For example, a water type move that is super effective against a fire type will instead be resisted.
- Broken cup; 6 random Pokémon will be assigned with random typings and random moves. There will also be an extended table of moves to choose from for more variety.

The data for this game will be held in JSON format and will contain all the information about every Pokémon like its type and possible moves, along with a list of all moves and its effects. This will make it simple to add and remove Pokémon after compilation and won't rely on hardcoding specific effects. The player will also be able to create their own JSON file that contains data about a team they would like to create for use within the standard game mode

2. List of Classes

- **Window**: represents a window for the Pokemon Simulator game and has a Vector2 size attribute and functions like Open(), Close() and get_window_size().
- **Renderable** *(abstract base class "renderable" for simple rendering)*: represents a game object and has attributes like sprite, position and size
- **Sprites** (***on heap*** *to hold all the textures; public getters but no setters)*: represents a collection of sprites and has functions like add_sprite and get_sprite.
- **Game Container**: represents the Game Container of the Pokemon Game and has attributes likehp_index and background_index.
- **Main Menu**: has a draw function that can display the Main Menu.
- **Gamestate**: represents Game state for the player and inherits from the Window and Renderable class. It also has an additional current_turn attribute and a swap_move() function that can swap moves for the current Player's Pokemon.
- **Player**: represents a player of the Pokemon game and inherits from the Renderable class. It has an additional attribute like current_Pokemon.
- **Bot**: represents the bot of the game and inherits from the Player class. It has an additional make_move() function to generate moves against the Player.
- **Pokemon**: represents a Pokemon of the game that contains all the attributes like name, type, base_stats, stats, max_hp and current_hp. It also has functions like take_damage(), modify_stats() and receive_move() to
- **Random_Pokemon**: inherits from the Pokemon class and has a generate_random() function to generate random pokemons for the player.
- **Move**: represents the moves for the Pokemon. It contains attributes like name, effect, type_one, type_two and a constructor Move(string, string, string, string) to create a move for the Pokemon.
- **Read_Pokemon_File**: inherits from the Player class and has a read_from_JSON() function to read Pokemon data from a JSON file.

3. Data and Function Members

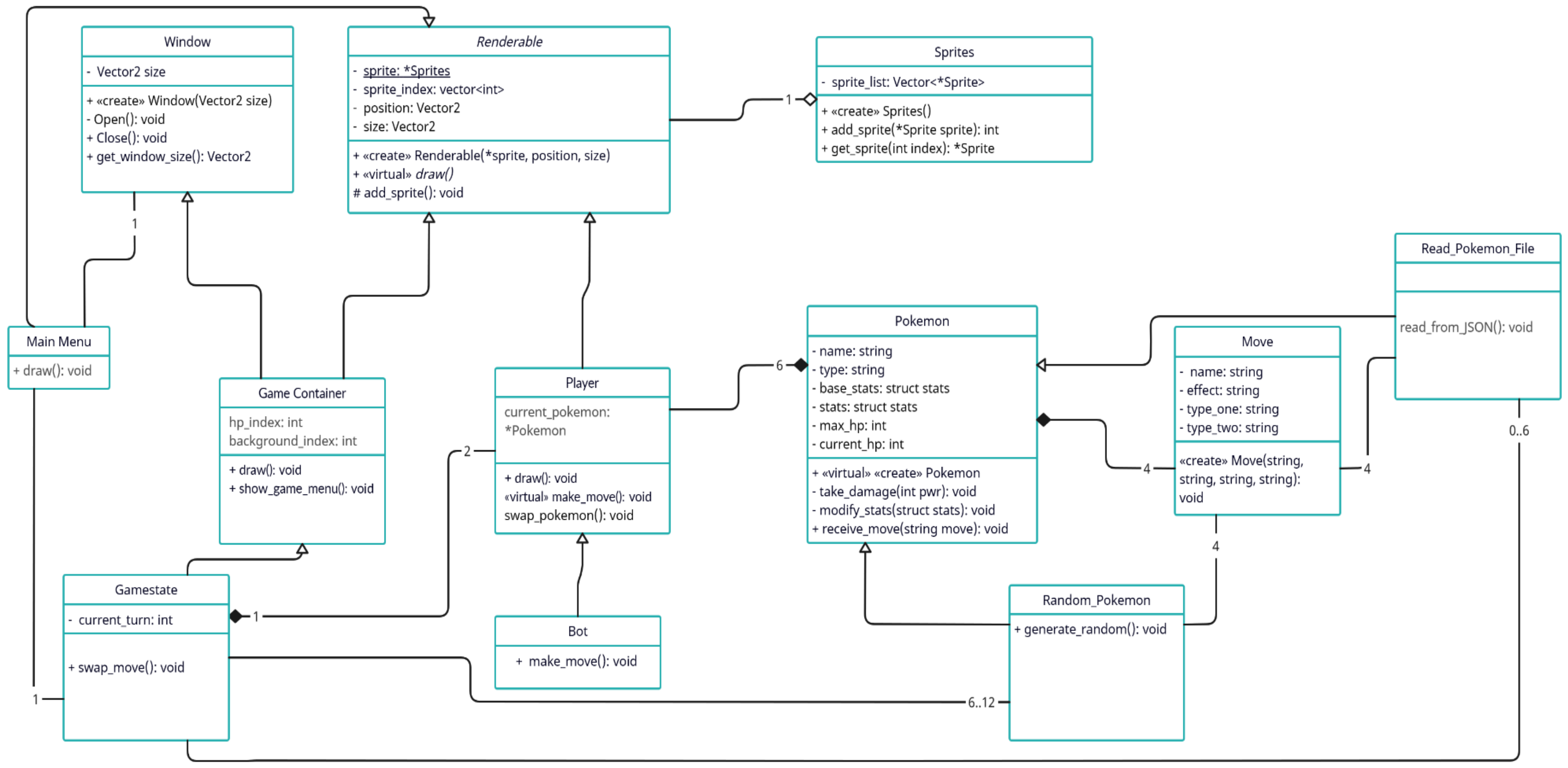

Figure 1: UML of Pokemon simulation

Window

- Attribute:
    - Vector2 size
- Function:
    - Window(Vector2 size): a constructor to create the game window
    - Open(): open window
    - Close(): close window
    - get_window_size(): returns the size of the window as Vector2

Renderable

- Attribute:
    - sprite: *Sprite
    - sprite_index: Vector<int>
    - position: Vector2
    - size: Vector2
- Function:
    - Renderable(*sprite, position, size): constructor to create a renderable object with a sprite, position and size
    - virtual draw(): a virtual function that can be overridden by the derived class to provide different implementation
    - add_sprite(): adds sprite to the Renderable

Sprite

- Attribute:
    - sprite_int: Vector<*Sprite>
- Function:
    - Sprite(): a default constructor to create a Sprite
    - add_sprite(*Sprite sprite): adds a sprite to the sprite object
    - get_sprite(int index): returns sprite at the specific index

Main Menu

- Function:

- draw(): draws the Main Menu screen

Game Container

- Attribute:
    - hp_index: int
    - background_index: int
- Function:
    - draw(): draws the Game Container
    - show_game_menu(): displays the game menu

Gamestate

- Attribute:
    - current_turn: int
- Function:
    - swap_move(): swaps move for the current Pokemon

Player

- Attribute:
    - current_pokemon: *Pokemon
- Function:
    - draw(): draws the Player's pokemon
    - virtual make_move(): a virtual function that can be overridden by the derived class to provide different implementation
    - swap_pokemon(): swaps the current Pokemon with another

Bot

- Function:
    - Make_move: makes a move for the bot

Pokemon

- Attribute:
    - name: string
    - type: string
    - base_stats: struct stats

  - stats: struct stats
  - max_hp: int
  - current_hp: int
- Function:
  - Pokemon(): a default constructor to create a Pokemon
  - take_damage(int pwr): reduces a Pokemon's current HP by a specified power value
  - modify_stats(struct stats): adjusts a Pokemon's stats
  - receive_move(string move): takes a move

Random_Pokemon

- Function:
  - generate_random(): generates random Pokemon

Move

- Attribute:
  - name: string
  - effect: string
  - type_one: string
  - type_two: string
- Function:
  - move(string, string, string, string): a constructor function to create moves for pokemon

Read_Pokemon_File

- Function:
  - read_from_JSON(): reads Pokemon data from a JSON file

4. Relationships between Classes

As seen in Figure 1, the parent class of Player, Game Container, and Main Menu is the Renderable class, which serves as an Abstract class that has a virtual function called Draw. The different attributes and functions will be inherited by all aforementioned classes for rendering the various parts of the game on screen.

Specifically, the Game Container class inherits from Renderable and Window class, and is the parent class of Gamestate class. Similarly, the Bot class inherits from the Player class and utilizes a virtual function through polymorphism to execute moves.

The Player class carries a compositional relationship with the Gamestate class, wherein two players correspond to a single Gamestate object, representing the bot and user. This relationship is present with the Pokemon and Player classes relationship, except with a six–pokemon-to-one-player ratio, due to the team format.

In a similar fashion, Sprites is a static class that uses the functionalities of Renderable, but exists independently. The connection exists due to several classes such as Player needing the Sprites class to render Player sprites.

The Read_Pokemon_File will inherit Pokemon as well as associate the Gamestate class in a one-to-many (up to six) relationship. The Pokemon data contained by the JSON files that are read by the class are utilized upon the creation of Pokemon objects. Additionally, it is associated with the Move class with four moves. This is due to the limit of four moves on each Pokemon and the information the Move class requires from the JSON file. The Moves class shares a four-to-one relationship with Pokemon (each Pokemon has four moves that the Player will command) as a compositional relationship and is associated with the Random_Pokemon class, which inherits from Pokemon. This relationship and class exists to cater for the randomized Pokemon needed in various modes. As these may require 6 or up to 12 random Pokemon (representing Bot and User team), the Gamestate class is associated with said class. Due to this, when the simulation starts, the Gamestate has the ability to call Random_Pokemon or Read_Pokemon_File to generate the needed Pokemon for the battle that will be passed to the Player.

5. Timeline

Week 9

The main focus for week 9 will be to construct a solid foundation for the code. The aim will be to construct clean, expandable code that allows new features to be efficiently implemented.

- Alex will complete sprite handling and drawing. Being a fundamental feature, a rudimentary version will be completed in 1 day and refined during the days following.
- JSON parsing and interpreting and will be implemented using the external library *JSONCPP*. This will be done by Alex which will take 2 days to create and an extra 1 day to handle edge cases and ensure large files are read correctly
- Game state management will be completed by Camille, including handling turn order, the player's active pokemon, and starting a game using data from a parsed JSON file or assigning random pokemon. This will take 4 days.
- Roselyn will work with Camille when implementing game state management and Alex when creating JSON parsing to create the Pokemon class. This is to ensure it communicates with the rest of the program dynamically . This will take 4 days.

Week 10

The creation of a minimum viable product will be attempted by the middle of week 10. This will allow for one and half weeks of refinement and testing. The normal mode, with both random and chosen teams, should be completed by this time.

- Unit tests and debugging for the normal mode will be engineered by Alex in two days.
- The next game mode, Broken Cup, with its function members, adjusted rule system, and user interface is to be implemented by Alex and Camille for the entire week.
- Mayhem, the last game mode, will be applied by Alex and Roselyn for the entire week.

Week 11

The final refinements and unit testing will be carried out on Week 11:

- User documentation is recorded alongside the entire process of the project by Roselyn.
- The finalization of the code, unit testing, and submission will be decided by all team members for one day.
- If time permits, the entire team will work together on implementing a simple LAN multiplayer mode with a chat room. Around 5 days of extra time will be required to complete this feature.

6. User Interaction Description

The user will be provided with the selection of several game modes: normal, broken cup, and mayhem with their own set of rules. Additionally, options to reload/delete a save file or an exit button for the simulation will be available. By clicking on the options using a mouse, the User will be able to select each of these options. The game will mostly use mouse clicks to select moves and options throughout the game. Normal mode will provide the options of a random team mode or a team-picker mode that will allow the choice of six Pokemon out of 150 on the user's team. Mayhem and broken cup remain strictly randomized for the user. For proper input by the user, error messages will be applied where necessary.

Loading into battle, sprites for the bots and users Pokemon will be displayed. Additionally, the user will have the choice of four moves owned by their Pokemon to battle a randomized bot team (random team for all modes). The user will also have the option to switch to another Pokemon. Statistics such as Health Points (HP) and Typing will be shown by a pop-up menu next to the Pokemon. The same pop-up menu will be displayed next to the bots Pokemon sprite. Every battle turn and action by the user and bot will be recorded in text form and displayed by a battle log next to the battle. Once all Pokemon of either team are defeated, the battle ends, and a pop-up of victory/defeat shows. The user will be able to exit the battle by an exit button at any point in time and be redirected back to the menu.

7. Unit testing and debugging plan

7.1. Testing

The program will be tested using a combination of unit testing and organized input/output testing.

The unit tests will cover all individual classes, functions and components to ensure that the program performs as expected.

- Player Class:
  - Test that a player's Pokemon are correctly initialized
  - Test that a player can switch between their Pokemon during a battle
  - Test that a player's Pokemon can execute moves correctly
- Pokemon Class:
  - Test that a Pokemon's stats and HP are correctly initialized
  - Test that a Pokemon's HP decreases when it takes damage
  - Test that a Pokemon's moves can be executed correctly
  - Test that a Pokemon's type and name are correctly stored
- etc.

The input/output tests will verify that the program reads and writes data correctly from the file and that the program handles user input and output correctly.

7.2. Debugging:

- Use a debugger to step through the code and identify where errors occur.
- Use logging to track the flow of the program and identify errors or bugs that occur during execution.
- Use error handling to catch and handle exceptions that occur during program execution.
- Test the program under different conditions, such as different input values or different player actions, to identify and fix errors that occur in specific situations.

Makefile and README files will be included in the project, which will provide clear instructions on how to compile, run and test the program. The Makefile will handle dependencies, and the debugging and release builds will be separated. The README file will explain how to use the program and will list any known issues or limitations of the program.