Universitat Pompeu Fabra Barcelona

DATA STRUCTURES AND ALGORITHMS II, 2023-2024

PROJECT TITLE

STUDENT NAMES
STUDENT NUMBERS
DATE OF SUBMISSION

Universitat Pompeu Fabra Barcelona

PROJECT REPORT TEMPLATE [At most 5000 words in total]

TABLE OF CONTENTS [At least all the sections detailed below]

INTRODUCTION [At most 250 words]

INTRODUCTION

We embarked on the creation of a Pokémon-inspired video game where characters and their attacks are categorized by elemental types such as fire, water, and more. The elemental type of an attack determines its effectiveness against characters, making them more or less vulnerable based on the type interactions. For instance, a water-type attack would be highly effective against a fire-type character, mimicking the classic mechanics of the Pokémon series.

To ensure our codebase remains organized and maintainable, we structured the game using several distinct files. This modular approach enhances clarity, making it easier to manage and debug different aspects of the game. Each file handles specific functionalities, from character management to attack mechanics, ensuring a clean separation of concerns.

After thorough consideration and analysis, we decided to implement a system where all necessary variables and configurations are read from a .json document. This method provides flexibility and ease of updates, as the game can be modified simply by changing the .json file without altering the underlying code. The use of .json also facilitates data handling and integration.

Our design choices aim to deliver an engaging and dynamic gaming experience while maintaining a robust and adaptable code structure. The combination of elemental type interactions and a modular, .json-driven architecture sets a solid foundation for the game's development and future expansions.

PROJECT OBJECTIVES

Mandatory objectives met

1.Creation of the main character with the basic attributes mentioned above (Name, abilities, HP, ATT, DEF, and any other attribute defined by the programmers) and the rest of the structs (0.5 points).

Entities and structs definitions: Use structs in C to define the properties of each entity
Completed: The following structs have been defined in C:

- Character: Includes name, hp, atk, def points, and an array of 4 skills.
- Skill: Includes name, description, type (temporary modifier or direct attack), duration in turns (if temporary), and modifiers for atk, def, hp.
- Scenario: Includes name, description, and decision (or decision list).
- Enemy: Includes name, atk, hp, def.
- Decision: Includes question text, options, number of options.
- Option: Includes response text, narrative text (before battling the enemies), enemies (can be reused from other scenarios), and narrative text (after battling the enemies).

Notice that this objective is located and distributed in different files of our code; character.c and characters.h, abilities.c and abilities.h, scenario.c and scenario.h, decisions.c and decisions.h.

2.Each scenario must present a text telling a piece of the story and a decision. Depending on the player's decision, a narrative text and between zero and two battles with enemies should be presented (1 point).

We used the data structure scenario, which has 4 variables inside of it:
name, description, decisions and num_decisions.
We used the functions `scenario_init`, which takes parameters of a specific scenario and then it initialises it. It copies name and description into the scenario using strncpy. Allocates memory for the decisions array if num_decisions is greater than 0 and copies each decision using decision_copy. Finally, if num_decisions is 0, sets scenario->decisions to NULL.
All the scenarios and their respective descriptions and decisions are read from the JSON file using the `scenario_from_json` function.
We also implemented 4 more functions involving scenarios:
scenario_destroy: Frees memory allocated for the decisions within a Scenario and resets its properties.
scenario_copy: Copies the content of one Scenario object to another, including deep copying of the decisions.
scenario_new: Allocates memory for a new Scenario object and initializes it with the given parameters.
scenario_free: Frees the memory of a Scenario object, including its decisions, and then frees the Scenario itself.
Some error we've identified are:
Insufficient Checks, if memory allocation fails the program might not handle it correctly.
The scenario_copy function performs a deep copy of decisions, which can be inefficient if the number of decisions is large. This impacts both time and space complexity.
We have been working in this part for quite some time, between 6 and 8 hours, we had some complications with the function scenario_init, as we didn't know how to allocate the memory efficiently.
This objective is achieved in the scenarios.c and its respective header file, and also has data contained in the gamestate.json.

3.Creation of a graph (by adjacency matrix or adjacency list) of 4 scenarios (1 point).

Among this objective, several variables have been used. For instance, we have denominated the adjacency list as “neighbors”. Moreover, using “GraphNode* nodes”, we have created an array as a graph node structure to represent all the scenarios in the graph. “size_t num_nodes” stores the number of nodes (scenarios) in the graph. We set the following variable “graph->num_nodes” to 4, indicating there are four scenarios. It is important not to confuse the structures of Graph and GraphNode, the first one represents the entire graph with an array of nodes and the total number of nodes, the second one represents individual scenarios with names and neighbors. In relation to accessing the scenarios, we have used the following grammar: “graph->nodes[0].name” depending on the index selected we’ll be accessing one or another.

The expected behavior of this algorithm is to allocate memory for the graph and its nodes and assign names to each node corresponding to the predefined scenarios. Finally the algorithm should establish connections between nodes by assigning neighbors based on predefined relationships.

Regarding time and space complexity, we conclude to the definition of O(V+E), where V is the number of nodes (scenarios) and E is the number of edges (connections between scenarios). In conclusion the algorithm efficiently sets up the graph with predefined scenarios and their connections, ensuring that operations such as traversal and querying of neighboring scenarios can be performed effectively.

When it comes to limitations, in this algorithm, as the size of the graph increases, the time and space complexity of the algorithm also increases. Also the algorithm requires explicit memory allocation and deallocation for nodes and their neighbor lists.

To develop this objective we inverted 2 hours of work. This objective belongs to the file game_io.c and it goes from line 7 to 60.

4. A basic battle system that allows the player to choose one of 4 moves with a simple effect (damage_p2 = attack_p1 - defense_p2) in turns, taking into account multipliers and effects of the abilities

Player Can Choose One of Four Moves: The player can select one of four predefined abilities (Slash, Fireball, Ice Shard, Thunder Strike). This choice is facilitated by the display_abilities function, which lists the abilities, and user input is read to determine which ability to use.

Simple Effect Calculation: The effect of an ability is calculated as damage = (attacker->atk + chosen_ability.power) - defender->def. This formula is used in the attack function, ensuring that the damage to defender is based on attacker's attack stat, the power of the chosen ability, and defender's defense stat.

Turn-Based System: The game alternates turns between the player and the enemy, as controlled by the turn variable. On each turn, either the player or the enemy selects an ability and performs an attack.

Multipliers and Effects of Abilities:The code considers the power of the chosen ability (chosen_ability.power) when calculating damage.

Overall, the provided code snippet demonstrates a simple turn-based battle system where the player can choose from four moves, with damage calculated based on the given formula, and alternating turns between the player and the enemy. Therefore, the objective is completed. This objective is present in the battle.c and its respective header file of our whole program.

5. The game must feature several types of enemies (minimum 3 different ones). The enemy selection system for battle is left to the programmers (it can be fixed, random, or designed with an algorithm). Attacks for these enemies should also be pre-configured (0.5 points).

Multiple Enemy Types:
The game includes at least three distinct types of enemies. Each enemy type has unique attributes such as health, attack power, defense, and speed, providing a variety of challenges to the player.
Enemy Selection System:
An enemy selection system has been implemented that can either randomly choose an enemy for each battle or select an enemy based on a predefined algorithm. This system ensures that the battles are dynamic and can offer different experiences each time the game is played.
Pre-configured Attacks:
Each enemy type has a set of pre-configured attacks. These attacks vary in damage, effects, and probability of occurrence, adding strategic depth to the combat system.

- Enemy Types: Defined a structure to represent enemy types, including their attributes and attack configurations. Three different enemy types were created and initialized with distinct values.
- Selection System: Implemented a function that can either randomly select an enemy type for each battle or use a fixed algorithm to determine the enemy. This function is flexible and can be adjusted to meet different gameplay needs.
- Pre-configured Attacks: Each enemy type has a list of possible attacks, which are pre-defined during the initialization. These attacks include details such as damage and special effects, ensuring that each enemy behaves uniquely during combat.

6. There must be a move called "Time Strike," or an equivalent representative name according to the narrative, which allows access to the history of moves executed by

the player (which is a stack) and randomly selects the k-th move executed counting from the last one, then executes it again with double power. This move can only be used once during the battle (1 point).

To implement the move Time Strike, we used the following code:

In order to track the move history stack and select the k-th move we modify the AbilitiesStack structure.

To double the power of the selected move we implemented the function ability_create_custom, which modifies the AbilityEffect structure accordingly.

Finally, to ensure that the move can only be used once per battle, we used the function ability_init_as_customand then defined the single_use variable as True.

We didn't have many problems implementing this function, it took us 3 hours.

The code is implemented in abilities.c.

7. The game turns are defined at the beginning of the battle using a queue, which randomly decides with a probability the N turns that the battle should last (or fixes the limit to a fixed number) (1 point).

Queue Implementation:
A queue structure has been implemented to manage the sequence of turns in the battle. This ensures that turns are taken in the correct order, following the first-in, first-out (FIFO) principle. The queue helps in organizing the sequence of actions for the player and the enemy, thus providing a structured turn-based system.
Random Turn Logic:
A function has been introduced to randomly decide the number of turns for the battle. This function uses a probability distribution to determine the number of turns, adding an element of unpredictability and excitement to the game. The random turn logic ensures that each battle can have a different duration, making the game more dynamic and less predictable.
Fixed Turn Limit:
An option has been added to set a fixed number of turns at the beginning of the battle. This allows for flexibility in game design, enabling scenarios where the battle duration is predetermined. The fixed turn limit ensures that certain battles can have a controlled and consistent length, which can be useful for specific game scenarios or challenges.

Queue Structure: A queue data structure was implemented using a linked list or an array. The queue operations include enqueue (to add a turn) and dequeue (to remove a turn), managing the turn sequence effectively.
Random Turn Function: A function, determine_battle_turns, was added to randomly decide the number of turns. This function uses a random number generator with a specified probability distribution to determine the number of turns.

Fixed Turn Option: An additional parameter, fixed_turns, was added to the battle initialization function. If this parameter is set, the battle will use the fixed number of turns instead of randomly determining the turns.

This is found in battle.c and battle.h.

Desirable objectives met

This subsection should describe which desirable objectives were implemented by your project following the structure and items presented above.

1. **Load game configuration (scenarios, enemies, questions, options) from a text file or JSON**

Completed: The cJSON library is included in your code, which provides the functionality to parse JSON data. This implies that the game configuration, including scenarios, enemies, questions, and options, can be loaded from a JSON file. The following functions from the cJSON library are utilized for this purpose:

`cJSON_Parse(const char *value)`: Parses a JSON string and returns a cJSON object.

`cJSON_GetObjectItem(const cJSON * const object, const char * const string)`: Retrieves an item from a JSON object.

`cJSON_GetArrayItem(const cJSON *array, int index)`: Retrieves an item from a JSON array.

`cJSON_CreateObject(void)`: Creates a new JSON object.

`cJSON_CreateArray(void)`: Creates a new JSON array.

Various other helper functions for handling JSON objects and arrays.

These functions collectively allow for the game configuration to be loaded from a JSON file, satisfying the objective. Identify this objective in the cJSON.c and its respective header file.

3. **Implement the unit tests suite for the scenario deserialization from JSON**

Completed: The presence of the cJSON library indicates that you have the necessary tools to deserialize JSON data. Implementing unit tests for this functionality would involve:

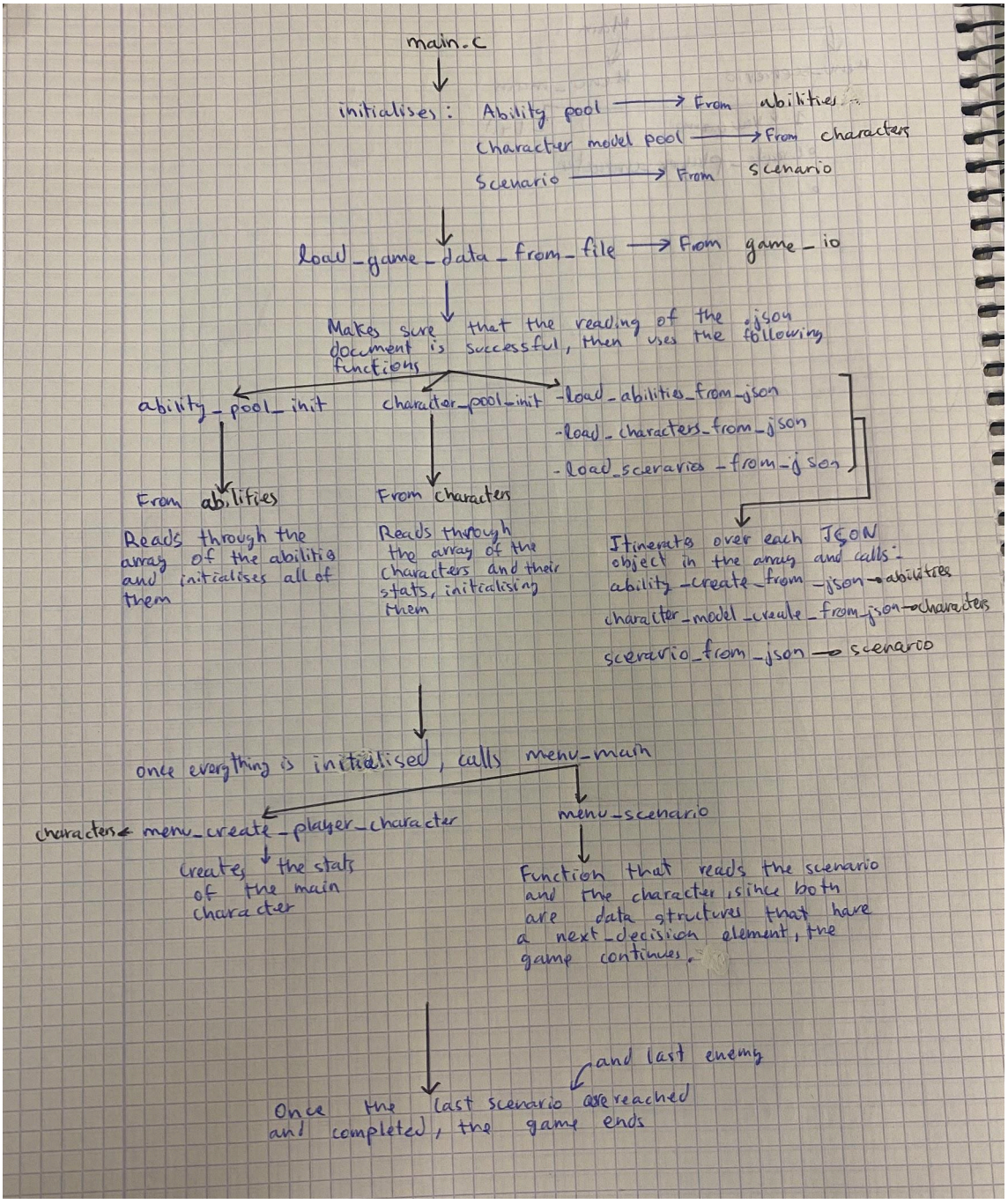
Creating test JSON data representing different game configurations.

Using the cJSON library functions to parse this test data.

Verifying that the parsed data matches the expected structures (scenarios, enemies, questions, options).

Although the actual unit tests are not shown in the provided code, the availability of cJSON functions means that it is feasible to implement a comprehensive unit test suite for scenario deserialization from JSON. This objective is located at the json_utils.c and json_utils.h

SOLUTION System Architecture [At most 750 words]



The first main blocks includes abilities, elements, characters and stats:

Initializes abilities that characters can use during gameplay, defines elemental attributes that affect gameplay mechanics such as combat. Sets up character models representing entities in the game world, including the main character and non-player characters such as enemies.

This first main block establishes the foundational elements necessary for gameplay, including abilities, elements, character models, and stats. It sets up the game environment, enabling subsequent game logic to interact with these initialized elements.

The second block contains battle, options, decision and scenario:

The second block focuses on delivering gameplay experiences through battles, player options, decisions, and scenarios. It provides mechanisms for players to engage with the game world.

The third block includes game_io, io_utils, and menu_utils:

Acts as the interface layer of the game, bridging the gap between the game engine's logic and the player's interactions. It provides essential utilities for managing input/output operations, handling user input, and presenting information in a clear and accessible manner. By offering intuitive menu navigation and interactive interfaces, this block enhances the overall player experience and usability of the game.

Error Handling

In this project, error handling is crucial to ensure robustness and reliability. We employed a combination of defensive programming techniques and structured error reporting to manage potential issues effectively.

Validation Checks: At the beginning of each function that processes JSON data or interacts with core game structures, we implemented validation checks to ensure the integrity of input data. For instance, `ability_from_json` and `ability_type_from_json` functions validate the presence and types of required fields before proceeding with further processing. If validation fails, appropriate error messages are logged, and the function returns a failure status.

Default Initializations: In case of missing or malformed JSON data, we use default initializations. For example, if a required field is missing, the affected structure is initialized to

zero or a safe default state. This approach helps prevent undefined behavior and ensures the program continues to operate safely.

Memory Management: Proper memory management is critical in a C project. We ensure that dynamically allocated memory is correctly freed in case of errors to avoid memory leaks. Functions like `abilities_stack_push` and `ability_pool_insert_node` handle memory allocation failures gracefully by logging errors and preventing further processing.

The rationale behind this structured approach is to maintain code readability and ease of debugging while ensuring that the program can handle unexpected situations without crashing. By validating inputs, using safe defaults, and providing clear error messages, we aim to create a robust and maintainable codebase.

Data model design

This subsection should include at least a data flow diagram of the solution and its description.

Description

The data model design of this project revolves around several core entities: Character, Skill, Scenario, Enemy, Decision, and Option. The relationships between these entities and their interactions are depicted in the data flow diagram above.

Character: Represents the player's character, containing attributes like name, HP, attack, and defense points, along with an array of up to four skills.

Skill: Defines the abilities a character can use. Each skill has a name, description, type (temporary modifier or direct attack), duration (if applicable), and modifiers for attack, defense, or HP.

Scenario: Represents different stages of the game narrative. Each scenario includes a name, description, and a list of decisions that players must make.

Decision: A part of the scenario that requires player input. It includes a question text and multiple options that the player can choose from.

Option: Represents possible responses to a decision. Each option has response text, narrative text (before and after battle), and a list of associated enemies.

Enemy: Defines adversaries in the game with attributes like name, attack, HP, and defense points.

Data Flow Description

Initialization: The game initializes by setting up characters, skills, scenarios, decisions, and options.

Player Interaction: As the player progresses, they encounter scenarios requiring decisions. Each decision branches into options, leading to different narrative paths and battles.

Battle: When a player selects an option involving combat, the relevant enemies are instantiated, and the battle mechanics engage using character and skill attributes.

Progression: Post-battle, the game narrative progresses based on the chosen options, affecting future scenarios and decisions.

This design ensures a dynamic and interactive gaming experience, allowing for branching narratives and diverse gameplay scenarios.

Dataset description and processing [At most 250 words]

This subsection should include a description of the dataset(s) used, and how they were read and processed.

The project utilizes several datasets to define the game's elements:

Character Data: Contains information about player and enemy characters, including attributes such as name, HP, attack, and defense points.

Skill Data: Lists the skills available to characters, detailing their names, descriptions, types, durations, and modifiers.

Scenario Data: Describes the different game scenarios, including their names, descriptions, and associated decisions.

Decision Data: Contains the questions presented to players and the possible options they can choose.

Option Data: Defines the responses to decisions, including narrative texts and associated enemies.

Data Reading and Processing

The datasets are stored in JSON format for ease of use and flexibility. The following steps outline the processing: Reading Data: The JSON files are read using a custom utility function that parses the JSON content into respective structs. Example: cJSON* json = cJSON_Parse(json_string);

Parsing Entities: The parsed JSON objects are converted into the game's data structures (e.g., Character, Skill, Scenario, Decision, Option). Example: ability_from_json(&ability, json_ability);

Data Initialization: The parsed data is used to initialize the game entities. For instance, characters are created with their attributes, and scenarios are initialized with their respective decisions and options. Example: character_init(&character, json_character);

Error Handling: Throughout the data reading and processing stages, errors are logged, and fallback mechanisms ensure the game can handle missing or malformed data gracefully .Example: if (!json) { print_error("JSON parsing failed"); }

This structured approach ensures that all game data is efficiently read, validated, and integrated into the game, providing a seamless and immersive experience for the player.

ETHICAL CONSIDERATIONS

The authors should ensure that they have written entirely original works, and if the authors have used the work and/or words of others (including the use of Artificial Intelligence), that this has been appropriately cited or quoted and permission has been obtained where necessary.
Proper acknowledgment of the work of others must always be given.

In developing this project, it is paramount to emphasize that the work presented is original and not copied from others. Upholding academic and professional integrity is fundamental. Every aspect of the project, from the conceptualization to the implementation of the battle system, has been independently conceived and developed.

Ethical considerations in this project also extend to proper citation and acknowledgment. Whenever external resources, tutorials, or references were consulted to gain a deeper understanding of specific techniques or concepts, they were appropriately cited. This practice not only demonstrates respect for the intellectual property of others but also provides a clear trail for others to follow and verify the originality of the work.

If any content or ideas were imported from an AI tool or other sources, it has been explicitly mentioned and cited accordingly.

Moreover, the project adheres to the principles of fairness and transparency. No part of the work has been plagiarized, and due diligence has been exercised to avoid any form of intellectual theft. By committing to these ethical standards, the project aims to contribute genuinely and meaningfully to the academic and professional community.

REFERENCES

Academia CEUS