DATA STRUCTURES AND ALGORITHMS II, 2023-2024

EDA STRIKES AGAIN

Victor Brull (U232597)

Josep Cardona (U231747)

Jan Balayla (U233694)

Oriol Carreras (U232393)

STUDENT NUMBERS

29/05/24

ÍNDEX

1. INTRODUCTION

After much deliberation on how we wanted to set the atmosphere of the game and its structure, we decided to go for a classic RPG.

The game consists of five scenarios (plus the introduction) where the player must face all sorts of enemies and uncover the secrets of their past. The story is linear, meaning that upon completing one scenario, the player progresses to the next without any choice. However, in each scene, crucial decisions must be made to avoid enemies. For example, in the fourth scenario, there are a series of challenges and puzzles that the player must solve correctly in order to advance to the next trial without being attacked.

Moreover, the combat system of the game is based on random turns. Using a queue we can define in which turns our character will attack. The abilities of the combatants also have unique characteristics that the player will discover. Our game blends its story elements with battles against the protagonist's enemies.

Throughout this report, we will refer to the various problems we encountered and how we managed to solve them. The main issues relate to the loading of data structures for the scenarios, abilities, and enemies. Additionally, the "fight" function has been quite a challenge.

This report covers the project objectives, solution developed, system architecture, error handling, data model design, dataset description and processing, and ethical considerations. Each section provides detailed insights into the development process and implementation.

2. PROJECT OBJECTIVES

2.1. Mandatory Objectives

1. **Creation of the main character with the basic attributes and the rest of the structs:**

   Time required: 3h

   First of all, regarding character creation, we have implemented some data structures in our file structures.h, one of which pertains to the main character (explained later).

   For the implementation of the main character’s creation, we have implemented a function called story_character_creation() in structures.c which takes as a parameter a pointer to the Session struct and returns a Character struct. This function is responsible for creating a character in a game by allowing the player to input their character's name, select equipment, and choose 4 skills among 11. It is only run once per game since we only call it once in a function called new_game() in main.c.

On the other hand, we have implemented a total of 10 data structs:

1. **Skills struct**: represents a skill in the game, detailing its name, description, effect, type, and the modifications it applies to health, attack, and defence stats:

```c
typedef struct Skills{
    char name[MAX_NAME]; // name of the skill
    char description[MAX_LENGTH]; // brief description of the skill
    char effect[MAX_LENGTH]; // effect the skill has in the game
    int type; // type of skill, 0 or 1; 0 → not a soul skill, 1 → soul skill
    float hp_mod; // modifier that recovers character's HP when the skill is used.
    float atk_mod; // modifier that increases, in that turn, character's ATK when the skill is used
    float def_mod; // modifier that increases, in that battle, character's DEF when the skill is used.
}Skills;
```

2. **Enemy struct**: represents an enemy in the game, including its name, stats (health, attack, defence), number of turns it can take, skills it possesses, a skill multiplier, and its maximum health:

```c
typedef struct Enemy{
    char name[MAX_NAME]; // enemy's name
    int hp; // enemy's health points
    int atk; // enemy's attack points
    int def; // enemy's defence points
    int turns; // maximum number of turns the main character has to beat that enemy
    Skills skills[PLAYER_SKILLS]; // enemy's skills
    float multiplier_skill; // multiplier for the enemy's skills (0.5 or 0.75 or 1)
    int max_hp; // maximum enemy's health points
}Enemy;
```

3. **Option struct**: represents a choice available to the player, including the response text, narrative texts before and after a battle, and the enemies involved in this option:

```c
typedef struct Option{
    char response[MAX_LENGTH]; // text response for this option
    char previous_narrative[MAX_LENGTH]; // narrative text shown before battling enemies
    char after_narrative[MAX_LENGTH]; // narrative text shown after battling enemies
    Enemy* enemies; // array of enemies involved in this option
    int en_num; // number of enemies involved in this option
}Option;
```

4. **Decision struct**: represents a decision point in the game, containing the question text presented to the player and the list of possible options they can choose from:

```c
typedef struct Decision{
    char question_text[MAX_LENGTH]; // question text for that decision
    int options_number; // number of options available for that decision
    Option* options; // array of options for that decision
}Decision;
```

5. **Scenario struct**: represents a scenario in the game, including its name, description, completion status, pointers to the next and previous scenarios in a linked list, a list of decisions available in this scenario and its corresponding number (is going to be explained more in detail in the third mandatory objective):

```c
typedef struct Scenario{
    char name[MAX_NAME]; // scenario's name
    char description[MAX_LENGTH]; // scenario's description
    int completed; // 0 → not completed, 1 → completed.
    struct Scenario *Next; // next scenario
    struct Scenario *Previous; // previous scenario
    Decision* decisions; // array of decisions in this scenario
    int dec_num; // total of decisions in that scenario
    int ID; // scenario's number (scenario 1 → ID = 0)
}Scenario;
```

6. **Character struct**: represents the player's character, including its name, stats (health, attack, defence, velocity, soul), and an array of 4 skills the character possesses.

   We must highlight that the velocity stat is unique to the character and it increases its chances to make a move in a turn up to 10 points more, it basically sums the 10% of the velocity points. Thus, if for instance the main character has 80 points of velocity, its chances to make a move in a turn will be 50% + (10% of 80) = 50% + 8% = 58% and, therefore, the enemy's chances will decrease up to 42%.

The character also has another unique stat called soul that is used for skills. There are some skills that are much more powerful than others, although they have a cost, 40 of soul. Hence, to utilise a soul skill, the player must have, at least, 40 of soul. 10 points of this special stat are generated each player turn.

```c
typedef struct Character{
    char name[MAX_NAME]; // character's name
    int hp; // character's health points
    int atk; // character's atack points
    int def; // character's defence points
    int vel; // character's velocity points
    int soul; // character's soul points
    Skills character_skills[PLAYER_SKILLS];  // character's skills
}Character;
```

(The following two structs are used for the skills dictionary and are going to be explained more in detail later on).

7. **HashNode struct**: represents a node in the hash table for storing skills, including a key, the skill name, the skill itself, the number of times the skill has been used, and a pointer to the next node:

8. **HashTable struct**: represents a hash table for efficiently storing and retrieving skills using hash keys, consisting of an array of pointers to HashNode and the size of the table:

9. **Session struct**: represents a game session, including the hash table of skills, the player's character, pointers to the current and first scenarios, current scenario's number and an array of enemies present in the session:

```c
typedef struct Session {
    HashTable* hash_skills; // hash table containing the skills
    Character player; // player's character
    Scenario *current_scenario; // current scenario
    int current_ID; // current scenario's number
    Scenario *first_Scenario; // first scenario in the list
    Enemy enemies[MAX_ENEMIES]; // array of enemies in the session
} Session;
```

10. **Intro struct**: represents the introductory text sections for the game, including text shown before and after entering the character's name, a prophecy text, and final introductory text:

```c
typedef struct Intro{
    char prev_name[MAX_LENGTH]; // introductory text shown before entering the character's name
    char after_name[MAX_LENGTH]; // text shown after entering the character's name
    char prophecy[MAX_LENGTH]; // prophecy text for the character
    char last[MAX_LENGTH]; // final introductory text
}Intro;
```

2. Story and player's decisions:

Time: 3h

The open_scenario function, located in stroy.c, starts by checking whether the current scenario has not been visited yet. If so, the function displays the scenario's name and description, presenting it alongside various decisions, each accompanied by multiple options for the player to choose from.

As the player makes choices, the selected option's narrative is displayed, providing context and depth to the decision-making process. If a battle is required, the function initiates combat sequences, displaying a message for each enemy encountered and calling the fight function to handle the combat. After any necessary battles, the post-battle narrative is displayed. Once all decisions and battles are processed, the scenario is marked as completed to prevent repetition in future playthroughs.

```c
void open_scenario(Scenario* scenario, Session* session){
    printf(CLEAR_SCREEN);
    save_session_to_file(session);
    printf("\n\n\n");
    printEnteringScenario(scenario->name);
    if(scenario->completed==0){
         //printf("%s\n",scenario->description);
         printf("\n");
         printWrapped(scenario->description);
        for(int i = 0; i<scenario->dec_num;i++){
            printf(BLUE UNDERLINE BOLD   "\n%s\n" RESET,scenario->decisions[i].question_text);
            for(int j = 0; j<scenario->decisions[i].options_number;j++){
                printf(GREEN BOLD  "%d. %s\n\n" RESET, j+1, scenario->decisions[i].options[j].response);
            }
            int option;
            printf("Enter your option: ");
            scanf("%d",&option);
            option--;
            //printf("\n%s\n",scenario->decisions[i].options[option].previous_narrative);
            printf("\n");
            printWrapped(scenario->decisions[i].options[option].previous_narrative);
            sleep(3); // 5-second delay
            printf(CLEAR_SCREEN);
            for(int j = 0; j<scenario->decisions[i].options[option].en_num;j++){
                printf(RED BOLD "\n//  BATTLE AGAINST %s  //\n" RESET, scenario->decisions[i].options[option].enemies[j].name);
                int win_num=fight(&session->player, scenario->decisions[i].options[option].enemies[j], session);
            }
            //printf("\n%s\n",scenario->decisions[i].options[option].after_narrative);
            printf("\n");
            printWrapped(scenario->decisions[i].options[option].after_narrative);
        }
        scenario->completed=1;
    }
    else{
        printf("You retrace your steps, only to hear a familiar voice echo in your mind: \"You've already been here.
        Why are we stopping? We can't afford to waste any time. Move forward and continue your journey.\"\n");}
}
```

3. Creation of 4 scenarios:

Time: 4h

Our game's history is completely linear and composed of 5 scenarios. This is initialised using a doubly linked list with the Scenario struct. Each scenario and its attributes are loaded with a function named scene_loader from a JSON file, scenarios.json. This function reads the file, extracts the information for each scenario, initialises the scenarios and their decisions, and finally links them together to form the game's timeline. Once all the scenarios are loaded and linked, the current session is then updated with the first scenario in the list, and memory is freed to deallocate. With this method, a doubly linked list is created where each Scenario struct contains a pointer to the next and previous scenarios in the list.

4. Battle system (damage_p2 = attack_p1 - defense_p2) by turns, taking into account multipliers and effects of the abilities:

Time: 10h

(All our battle functions are located in combat.c).

Our combat system for our RPG game is turn based, hence, players and enemies have turns to apply skills and make attacks. The use_skill function handles all arguments relevant to skill usage, such as attack, defence and damage multipliers and it is when calling apply_effects that modifies health, attack, defence or soul points. The level of damage caused by attacks will be determined by the function called deal_damage which takes into account the attacker's attack and defender's defence.

Turns are managed by the fight function using a queue (explained more in detail later on) influenced by character velocity. In every turn, the player selects any of his 4 skills, while enemies do it randomly. The battle continues until either the player or enemy's health reaches zero, determining the winner.

5. Type of enemies:

Time: 3h

We have a total of 8 different enemies every one with different stats and skills. As we have done with the scenarios, all the enemies data is loaded into the Enemy struct from a JSON file called 'presets.json' using a function called enemy_loader in structs.c. Their skills as well as the selection system for battle are pre-configured. Our different enemies are:

- Cave Minion
- Minion Guard
- Temple Minion
- Base General
- Warrior Spirit of the Grave
- Sacred Temple Protector
- Eternal Darkness Ascendant (EDA) (full life)
- Eternal Darkness Ascendant (EDA) (-35% HP)

6. Time Strike movement:

Time: 5h

Our fight function also initialises a stack where all the skills that the player uses during the battle will be loaded, enabling us to implement a exceptional skill that can only be used once per battle called Time Strike. What this does is select a skill from the stack randomly and use it, multiplying its effect by two.

Regarding the code, when this skill is selected, it first checks the number of skills in the stack. If it is not empty, it generates a random index k within the range of the stack size. Whereas it is empty, k is set to -1. Then, the get_kth_skill function selects the kth skill in the stack. Finally, if the get_kth_skill function has selected any, the use_skill function is called; it uses that skill with double power, and the time_strike_used flag is marked as used, setting it to 1. Nevertheless, if the get_kth_skill function has not selected any, it means that the stack is empty. Hence, the Time Strike skill cannot be used, and the player will be able to select another skill.

Time complexity of each stack function:

- push and pop: O(1)
- get_kth_skill: O(k)
- stack_size: O(n)

7. **Game turns based on a queue:**

Time: 5h

As we said before, the fight function is responsible for initialising a queue. Immediately, this is filled with the players and enemy turns, and, finally, the fight function determines whose turn it is dequeuing (the queue will be explained in more detail later on).

2.2. Desirable Objectives

<u>Data loading system:</u>

Time: 4h

With the intention of configuring the data structures of the main character, the enemies, the skills, and the scenarios, we have implemented a JSON format. In the code, there are two files named cJSON.c and cJSON.h that implement a series of functions used to extract information from a JSON file and load it to the structures wanted. There are three JSON files: extra.json saves the narrative for the introduction of the game; presets.json defines the data structures of the skills and enemies; scenarios.json saves the narrative of the story with the decisions, options, and enemies of each scenario.

Thus, we need to define three functions in charge of loading the data from the JSON files, which are introduction, skill_loader, enemy_loader, and scene_loader. The structure of the functions are basically the same. These functions are located in story.c and structures.c files.

These functions begin by opening the.json file and reading its contents into a string. After ensuring the file is properly read and closed, the function parses the JSON content using the cJSON_Parse function. If the parsing fails, it prints an error message and exits. The function then extracts the object array (scenario, enemy, skill) from the JSON object, ensuring it is valid and in the expected format.

Once the object array is validated, the function iterates over each element in the array. For each one, it extracts the data structure fields. At the end, it allocates memory for a new object structure and copies the extracted data into it. Nevertheless, details on file handling and error management will be discussed later in the report.

Finally, here we have the time complexity of each loader function:

- **skills_loader:** O(n+m+mplog(k)+m)
  - n: size of the file
  - m: number of skills
  - p: average number of attributes per skill
  - k: number of elements in the hash table

- **enemy_loader:** O(n+m+mp+mqlog(k)+m)
  - n: size of the file
  - m: number of enemies

- p: average number of attributes per enemy
- q: average number of skills per enemy
- k: number of skills in the hash table

- **scene_loader:** O(n+m+mp+mpq)
  - n: size of the file
  - m: number of scenarios
  - p: average number of decisions per scenario
  - q: average number of options per decision

<u>Queue:</u>

Time: 2h

For the combat system, we have implemented a queue in order to keep track of whose turn it is. On the one hand, let us define how the turns are computed.

Depending on the enemy's turns, we will initialise a queue with a length or with another. As it is said, our character has a stat of velocity. Using the fill_fight_queue function, we can determine how many turns our player will attack.

```c
void fill_fight_queue(Queue* q, int velocity, int turns) {
    int player_probability = 500 + velocity;
    srand(time(NULL)); // Seed the random number generator
    printf("Turn Queue Generated, Chance of player turn %0.1f%%\n", (player_probability) / 10.0);

    for (int i = 0; i < turns; i++) {
        int r = rand() % 1000;
        if (r < player_probability) {
            enqueue(q, 0); // Player's turn
        } else {
            enqueue(q, 1); // Enemy's turn
        }
    }
}
```

Depending on the probability of a random system, the function enqueues a player's turn (data node = 0) or an enemy's turn (data node = 1).

On the other hand, we need to define our queue system. Firstly, we define the queue, in which every node has data and a pointer pointing forward.

```
// Define a structure for the nodes of the queue
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Define a structure for the queue
typedef struct Queue {
    Node* front;
    Node* rear;
} Queue;
```

We create the queue using the createQueue function. Now, in the fill_fight_queue function, the turns can be added to the queue by enqueuing them once we know whose turn it is.

We enqueue a turn by adding a new element to the end of the queue. It first creates a new node with the given data. If the queue is empty, it initialises both the front and rear pointers to the new node. If the queue is not empty, it links the new node to the current rear of the queue and updates the rear pointer to the new node. This ensures that the new element is correctly added to the end of the queue while maintaining the queue's structure.

Thereafter, in the fight function, it dequeues the first element of the queue to check whose turn it is. Inversely to enqueueing, the dequeue function removes and returns the front element from the queue, updates the front pointer, handles empty queues, and frees the removed node's memory. As a result, the two functions would be defined like the picture.

Time complexity of enqueue and dequeue: O(1).

```c
// Function to add an item to the queue
void enqueue(Queue* q, int data) {
    Node* temp = newNode(data);
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
}

// Function to remove an item from the queue
int dequeue(Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return -1;
    }
    Node* temp = q->front;
    int data = temp->data;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    return data;
}
```

Dictionary:

Time: 2h

Furthermore, we have implemented a dictionary to search for skills based on a hash table. The skill dictionary uses two main data structures named HashNode and HashTable.

```c
//Structures for the dictionary of skills
typedef struct HashNode {
    char key[MAX_NAME];  // Stores the skill name
    Skills skill; // Stores the skill details
    int uses;   // Tracks how often the skill has been used
    struct HashNode* next;  // Pointer to the next node
} HashNode;

typedef struct HashTable {
    HashNode* table[MAX_SKILLS];  // Array of pointers to HashNode,
                                  // representing the hash table
    int size;   // Tracks the number of elements currently in the hash table
} HashTable;
```

The hash function is crucial for mapping skill names to specific indices in the hash table. It takes a string as input and returns an unsigned integer, previously ensuring that the hash value fits within the bounds of the hash table array. The function works by iterating through each character in the string, updating the hash value using a left bitwise shift and addition.

In order to create the table, we define a function (create_table_skills) that initialises a new hash function by allocating memory and setting all the table elements to null. Additionally, we have implemented insert_skill, find_skill and delete_skill. Let us clarify their functionality.

```c
void insert_skill(HashTable* hashTable, Skills skill) {
    unsigned int index = hash(skill.name);
    HashNode* newNode = (HashNode*)malloc(sizeof(HashNode));
    strcpy(newNode->key, skill.name);
    newNode->skill = skill;
    newNode->uses=0;
    newNode->next = hashTable->table[index];
    hashTable->table[index] = newNode;
    hashTable->size++;
}

Skills* find_skill(HashTable* hashTable, char* name) {
    unsigned int index = hash(name);
    HashNode* node = hashTable->table[index];
    while (node != NULL) {
        if (strcmp(node->key, name) == 0) {
            return &node->skill;
        }
        node = node->next;
    }
    return NULL;
}

void delete_skill(HashTable* hashTable, char* name) {
    unsigned int index = hash(name);
    HashNode* node = hashTable->table[index];
    HashNode* prev = NULL;
    while (node != NULL && strcmp(node->key, name) != 0) {
        prev = node;
        node = node->next;
    }
    if (node == NULL) return; // Not found
    if (prev == NULL) {
        hashTable->table[index] = node->next;
    } else {
        prev->next = node->next;
    }
    free(node);
    hashTable->size--;
}
```

- insert_skill: it calculates the index using the hash function. Then, creates a new HashNode, initialising its fields. Finally, it inserts the new node at the beginning of the linked list at the calculated index. Time complexity:
    - Best case: O(1)
    - Average case: O(1)
    - Worst case: O(n)
- find_skill: calculates the index of the element given using the hash function and traverses the linked list at that index, comparing keys until it finds the skill and returns it. Otherwise, it returns null. Time complexity:
    - Best case: O(1)
    - Average case: O(1)
    - Worst case: O(n)
- delete_skill: calculates the index of the element given using the hash function and traverses the linked list at that index. If the skill is found, it adjusts pointers to remove the skill from the list. Lastly, frees up the memory allocated for the node. Time complexity:
    - Best case: O(1)
    - Average case: O(1)
    - Worst case: O(n)

<u>Skill tracking:</u>

Time: 1h

Additionally, we have implemented a skill tracking which counts how many times the character has used that movement. When the user uses a valid skill, the usage count of a skill is incremented. The function player_turn, in combat.c file, calls find_node function which finds the skill node in the hash table, then increments the uses counter and prints the updated count. Indeed, the find_node function locates the skill node by its name in the hash table.

```
    } else if (skill_idx >= 1 && skill_idx <= 4) { // valid skill
        use_skill(player, enemy, &player->character_skills[skill_idx - 1], 0);
        push(skill_stack, &player->character_skills[skill_idx - 1]); // Push used skill onto stack
        HashNode* skill_node = find_node(session->hash_skills, player->character_skills[skill_idx - 1].name);
        skill_node->uses++;
        printf("%s has been used %d times.\n\n", skill_node->skill.name, skill_node->uses);
    }
}
```

This piece of code belongs to the player_turn function.

<u>Unit test:</u>

Time: 3h

As a desirable objective, we have executed a unit test suite to check some functionalities. Those functions serve to validate the game's key components and ensure a proper game experience. These tests encompass loading JSON data, combat mechanics and scenarios interactions. In order to go testing, we should enter the integer "TEST_CODE" defined in the main.h file. The unit test is located in the testsuit.c file. The test_menu function serves as the main interface, allowing us to choose which aspect of the game to test. Let us clarify what each test does.

Each testing function utilises the Session structure to manage and store game data. The test_json function initialises the session's hash table and loads skills, enemies, and scenarios from the JSON files. Finally, it prints them to the console for verification. Moreover, the test_combat function sets up a player character, allows enemy selection and sorting, and initiates a fight to test combat logic. At the last test, the scenario_test function lets us choose a scenario, opening and interacting with it.

Quick sort algorithm:

Time: 1h

Additionally, we have included a quick sort algorithm that sorts the enemies in the combat unit test by their health points, attack, or defence. The algorithm is based on that one seen in the theory classes. The test_combat function in the testsuit.c file calls quicksort in order to sort the enemies.

Time complexity:

- Best case: O(n*log(n))
- Average case: O(n*log(n))
- Worst case: O(n^2)

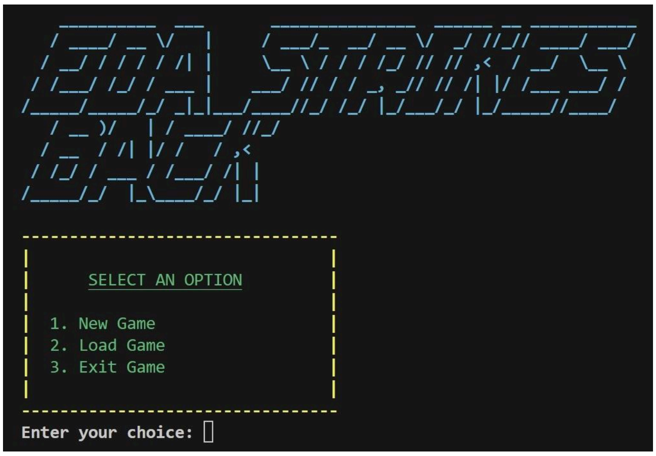
2.3. Optional Objectives

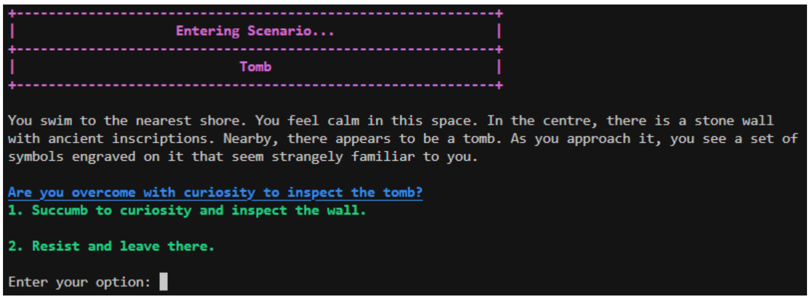
Game's interface:

Time: 3h

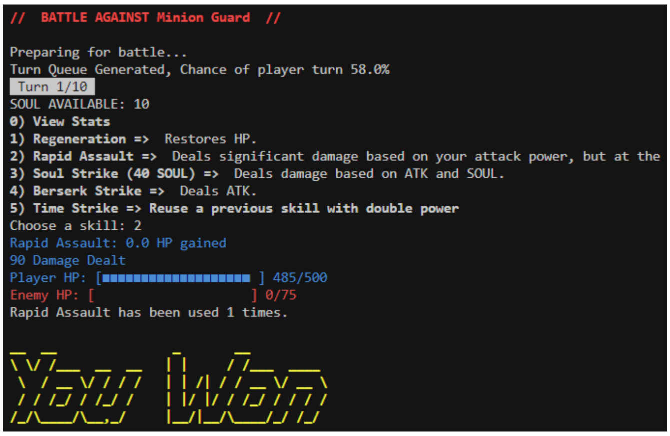
In order to improve the experience of the user in our game,. we have accomplished the optional objective of enhancing the game's interface. Here are some pictures to show the interface.

Firstly, we have this interactive menu, which welcomes the player.





We have coloured the scene by emphasising the question text and the possible decisions that the user can make.



Picture of winning a battle. When the user has to battle, it will display this screen in which there are the characteristics of the battle such as the skills with their definitions, the soul available, the number of turns and a progress bar of the fighter's life. After the combat, whether the player wins or loses, it will display a message.

In order to colour the screen, we have defined some “ANSI” colours in the structures.h file. Now, we can change the colour by writing the name of the colour wanted before the message. Also, we can reset to the original one by writing “RESET” whenever.

Furthermore, we have implemented the time.h library in which is included the sleep function. By passing an integer, the screen freezes the seconds of that integer. Moreover, in structures.h we have defined “CLEAR_SCREEN” that depending on the computer system, the screen is clear.

System for saving and loading:

Time: 3h

Moreover, we have implemented some functions, located at the main.c file, to save characters’ characteristics into a JSON file and to load the data structures from this JSON file.

For instance, the save_character function saves the character's data into a JSON object. It creates a JSON object and adds character data structures. It then creates a JSON array for the character's skills, iterating over the skills and adding each skill's name and usage count to the array. Given that, each skill is represented as a JSON object within the array. The expected return is the completed character JSON object.

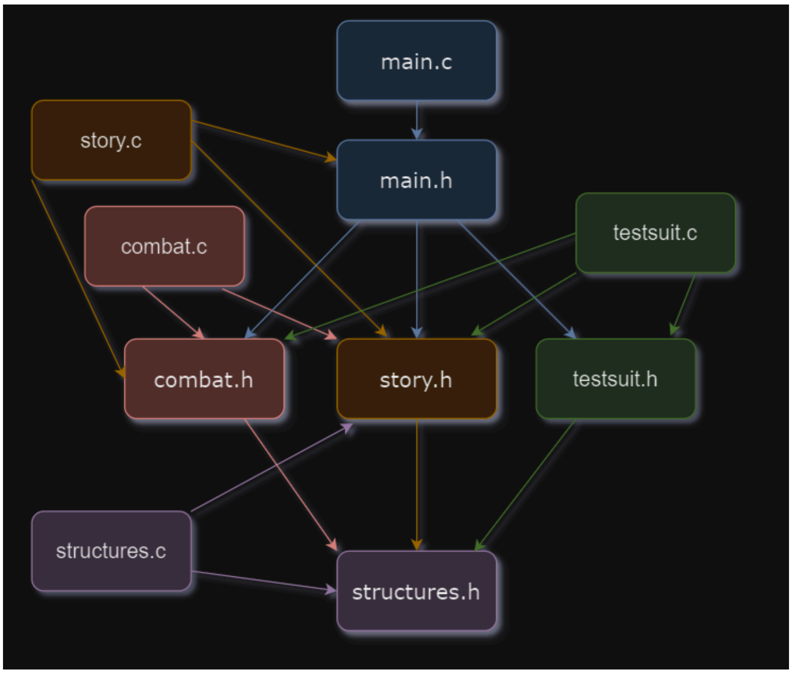
On top of that, the save_session function saves the entire session, including the player character and the current session ID. It creates a JSON object for the session, adds the character JSON object created by save_character, and includes the current session ID. This JSON object is then returned. Thereafter, the save_session_to_file function takes this JSON object, converts it to a string, and writes it to a file named session.json.

Alternatively, if we want to load the data from the session.json file, the load_session function is called. This function works similarly as the other load file functions. It extracts the data from the JSON file and assigns it to the character structure. Besides, it also retrieves the character's skills from the JSON array, finds the corresponding skill in the hash table, updates its usage count, and assigns the skill to the player's skills array.

Nonetheless, the program knows in which scene the user loads the game by the ID of each one of them. Each scenario has an ID number, so after loading the session data, the function also returns the session ID and sets the current_scenario pointer to the appropriate scenario based on this integer. It iterates through the linked list of scenarios, marking completed scenarios and finding the one matching the current session ID. Then, we can continue the game where the player left off.

3. SOLUTION

3.1. Structure Architecture



As we can see in the diagram, the structure of the game is based on 5 blocks. Each block performs a task for the perfect running of the game. To start with, the main block is in charge of starting the game, so we run the code by running the main.c file, which prints the menu and starts the game. Thereafter, it is able to save the current state and load it after. The main.c file includes main.h, which defines the functions of the previous file and the possible options that have the user in the menu, apart from the number of charging the unit test.

Moreover, the story block is in charge of getting the data structure for the introduction from the extra.json file. Nevertheless, the introduction function is called in another block. Continuing with the story block, it is assigned the open_scenario function that by passing to it a scene it prints its description and the possible decisions, also checks the enemies of that decision and calls the fight function.

On the other side, the combat block, as the name indicates, performs the gameplay related to the battles. Therefore, functions such as fight, player_turn or use_skill are defined and initialised in this block. It is worth mentioning that, combat block carries out stack and queues labours, it creates the queue in which the turns of the battle are defined and with the help of the stack it is capable of keeping track of the last skills of the actual combat to perform the time strike movement.

Additionally, the test suite block is in charge of testing functions of loading the JSON structures, checking any enemy in combat and initialising the game in the scenario desired. This block is very useful to us to quickly check any state of the RPG, also we can balance the stats in combat of skills, enemies and the character itself.

Finally, we have the structures block, which is included in all the previous blocks mentioned.

3.2. Error Handling

When we talk about error handling, we refer to the program's architecture for detecting errors, such as opening a document or filling a structure, and ensuring they do not cause a general failure in the code when running it.

In our RPG, we have made a concerted effort to detect any possible errors. On the one hand, when we ask the user for any option via the terminal, we ensure they provide a valid option, even though they enter an invalid integer or a character, thanks to our function called input_integer.

```c
int input_integer(char* message, int min, int max){
    int num;
    char term;
    printf(YELLOW BOLD"\n%s"RESET, message);
    while (1) {
        if (scanf("%d%c", &num, &term) != 2 || term != '\n') {
            printf(RED BOLD"\nEnter a valid integer:"RESET);
            while ((getchar()) != '\n'); // Clear the input buffer
        }else if (num < min || num > max) {
            printf(RED BOLD"\nEnter an integer between %d and %d:"RESET, min, max);
        } else {
            break; // Valid integer within range
        }
    }

    return num;
}
```

On the other hand, we have focused most on the JSON files. Let us focus on the load_session function in the main.c file that implements several error handling mechanisms to ensure robustness. However, this architecture is also implemented in the rest of functions that work with a JSON file.

```c
FILE *fp = fopen("session.json", "r");

if (fp == NULL) {
    printf("ERROR: Could not open file.\n");
    return;
}
```

- File opening:

If the file session.json cannot be opened, an error message is printed, and the function returns immediately.

```c
// Allocate memory to read the file
char* fileContent = (char*)malloc(fileSize + 1);
if (fileContent == NULL) {
    printf("Memory allocation failed\n");
    fclose(fp);
    return;
}
```

- Memory allocation:

After determining the file size, memory is allocated to read the file content. If memory allocation fails, the file is closed, and the function returns.

```c
cJSON* json = cJSON_Parse(fileContent);
free(fileContent);

if (json == NULL) {
    printf("Error parsing JSON\n");
    return;
}
```

- JSON parsing:

The JSON data is parsed from the file content. If parsing fails the function returns.

```c
if(session->current_scenario==NULL){
    printf(RED"\nScenario NOT found\n");
    return;
}
cJSON_Delete(json);
```

- Object not found:

While loading skills or the scenario, if this object is not found, an error message is printed, and the function returns.

3.3. Data Model Design

The game starts with an introduction, in which the player selects light or heavy clothing, and then begins the adventure. In the Cave Base, players can wait until the storm subsides (in which case the Cave Minion is encountered) or delve deeper into the cave, although both options lead to the base. In the base, the player can explore the path to the left or examine the machines. In both paths, the player meets Guard Minions, and the base is alerted. Later, the player can flee (in which the Base General is encountered) or save the souls (in which Cave Minions and Base General are encountered).

In the Tomb, the player can examine the wall or exit, both encountering a Warrior Spirit and leading to the Village. There, players can explore the centre or take the alleys, although both options lead to the player getting kidnapped. To be released, players can cut the ties or provoke the guards (in which Guard Minions are encountered and lead to be released). Later, players can explore the village or discover the way to the temple (in which Guard Minions are encountered), leading both of the options to leave the village.

In the Temple, there are three riddles. The only important one is the final one since, if the player answers it correctly, he gets a key to avoid fighting with a Base General later on. Once in the Fortress, players can either encounter with EDA directly or sabotage energy sources which encounter two Guard Minions and then EDA (-35% HP). After succeeding in the Fortress, the trapped souls are liberated and the adventure ends.

3.4. Dataset Description and Processing

As explained before in detail, our game data is located in several json files (extra.json, presets.json, and scenarios.json). When a new game is started, the new_game function is called, and then, this one calls a function called load_config located in structures.c. The load_config function is responsible for creating a table of skills called create_table_skills function and load all the skills, enemies and scenes data into the structures calling skill_loader, enemy_loader and scene_loader functions located in structures.c as well. The main purpose of these functions is, as said before, to load all the information from the json files to the structs.

4. ETHICAL CONSIDERATIONS

In order to enhance the visualisation of our videogame, we have implemented some numbers defined in the structures.h file that change the colour or underline the messages printed in the terminal. This piece of code has been obtained from a GitHub forum[1].

We found a simple and well designed web page[2] to ensure us to perform the best implementation of the queue for the program. We did not copy code from that page, however we have been inspired.

We were having errors with stack overflow, so we searched on the Internet to solve the problem. We found this page with a proper explanation of stacks[3]. We did not copy code from there, we only needed some references to solve the errors.

We wanted to implement functions to load the data structures from the JSON files, given that it was a desirable objective we thought that with a JSON structure the architecture of the game would improve. Hence, we asked ChatGPT to guide us on it. It gave us the cJSON.c and cJSON.h files that define the functions needed to load information. Moreover, in order to do the functions in which we load this information to the data structure we were supported by it. From there, we learned how that works and its functionalities. Additionally, it helped us to understand and find the time complexity of some functions and algorithms.

5. REFERENCES

ANSI colours:

https://gist.github.com/JBlond/2fea43a3049b38287e5e9cefc87b2124

Queue:

https://www.geeksforgeeks.org/introduction-and-array-implementation-of-queue/amp/

Stack overflow:

https://www.geeksforgeeks.org/introduction-and-array-implementation-of-queue/amp/