University of Science
Vietnam National University, Ho Chi Minh City

PROJECT REPORT

THE MATCHING GAME

23CLC10

CSC10002 - Programming Techniques

STUDENTS

Bui Duong Duy Cuong – 23127033

Tran Manh Hung – 23127195

INSTRUCTORS

Bui Huy Thong

Nguyen Ngoc Thao

April 15, 2024

Abstract

In the rapidly evolving field of Information Technology, continual learning and adaptation are essential. It starts with basic tasks such as working with console applications, and this project has pushed us to elevate our efforts by creating a game.

We have developed a classic game known as The Matching Game, also popularly referred to as the Pikachu Puzzle Game. This game features a board divided into multiple cells, each containing a letter from the alphabet. Players are tasked with matching pairs of letters following specific patterns: I, L, U, or Z shapes. Once a pair is matched, it vanishes from the board. The game concludes when all pairs have been successfully matched.

This report aims to thoroughly analyze and detail the elements of the game. We will introduce the game, followed by a tutorial on how the game works, explaining all the features, functions, and libraries. The game is programmed in C++, and is compatible with the Visual Studio compiler. We do not use g++ or other compilers as they are quite complex and require additional time to study if one wishes to use them effectively.

After reading this report, the instructors will have a comprehensive overview of how the game operates and the significance of the coding involved.

Member Information

Full Name	ID
Bui Duong Duy Cuong	23127033
Tran Manh Hung	23127195

Table of Contents

List of Figure

Chapter 1

Brief Introduction

The Pikachu Puzzle Game is an incredibly famous game that has been widely popular since its launch. Over the years, various publishers have competed to produce many versions based on the old core concept. Regardless, through the years, this genre of game has remained popular and has become a cherished memory for many.

Figure 1 Pikachu Puzzule Game (Game 24h)

"The Matching Game" is also a version based on the idea of the Pikachu Puzzle Game. We aim to create a game with a simpler interface and minimize the features while still retaining the core elements. Simply put, we create a board with various sizes depending on the difficulty level chosen by the player. The board contains cells with different alphabetical characters. Players need to find matching letters, and the connecting path between these letters must follow a specific pattern such as U, I, L, or Z. Cells that meet this condition will disappear, and the background image will gradually be revealed. This report will clearly explain how we sequentially implemented the steps to effectively run the game on a console.

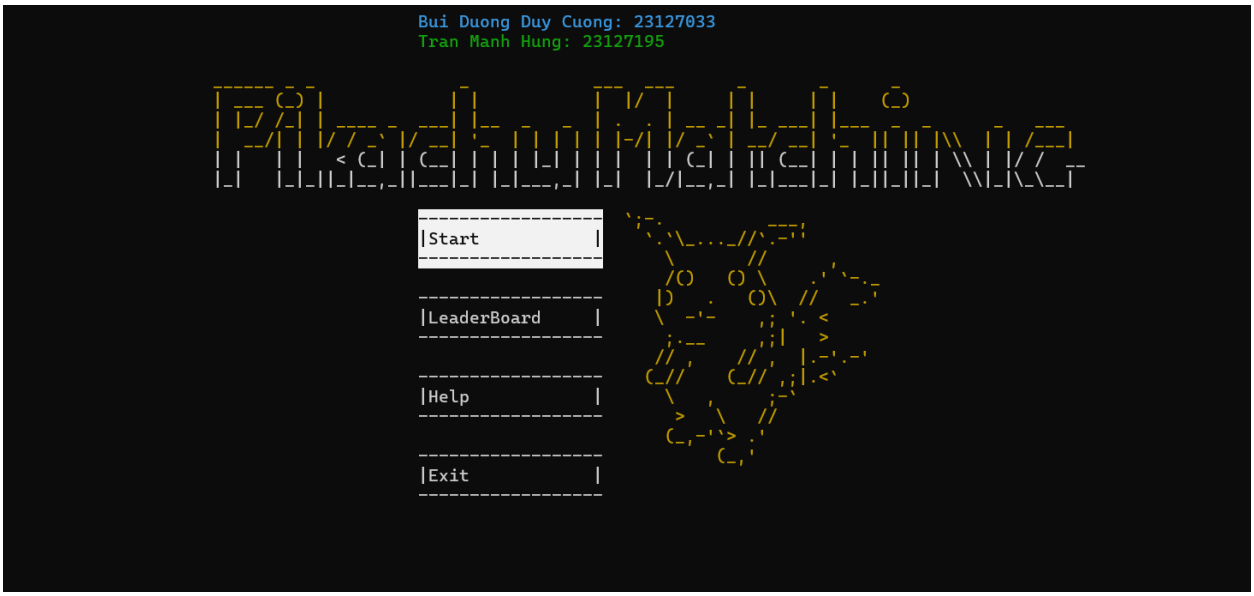

Figure 2 Our game - “The Matching Game”

This is the first time, after a learning process, that we have created and run a game on a console. Since everything is manually crafted from our own ideas, there may still be many shortcomings that we cannot fix in a short time. Therefore, we look forward to receiving your objective feedback on the issues so that we can gain more experience for the future.

Chapter 2
Game Tutorial

2.1 Case Diagram

Below is the Case Diagram of "The Matching Game", it describes how the game works in a summary and easy to visualize way.Note that this game is fully operated using KEYBOARD KEYS.

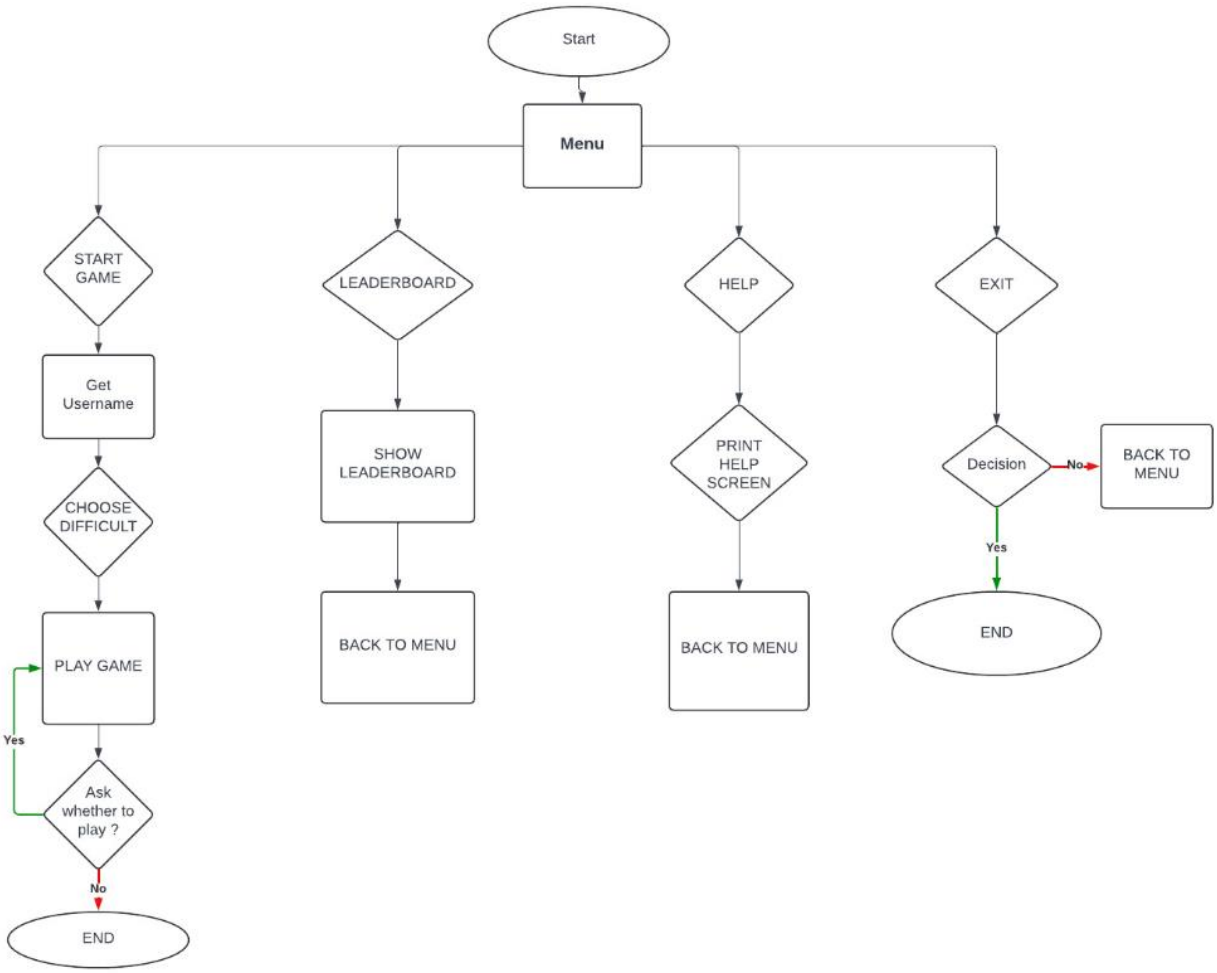

Figure 3 Case Diagram

2.2 Compile and Run

We have utilized the Visual Studio compiler for programming, making it the best compiler available for running this game. We have also created a demonstration video detailing how to download, install, and run the game flawlessly. Since the video file is too large, I will leave a YouTube link here for convenience, making it easier for you to view the video.

Please watch the video at: PIKACHU PROJECT | Demonstration Video by (Mạnh Hùng - Duy Cường) | HCMUS - Programming Techniques (youtube.com)

2.3 First look screen

Before going into the main game, be sure to **ENABLE FULLSCREEN** before playing.

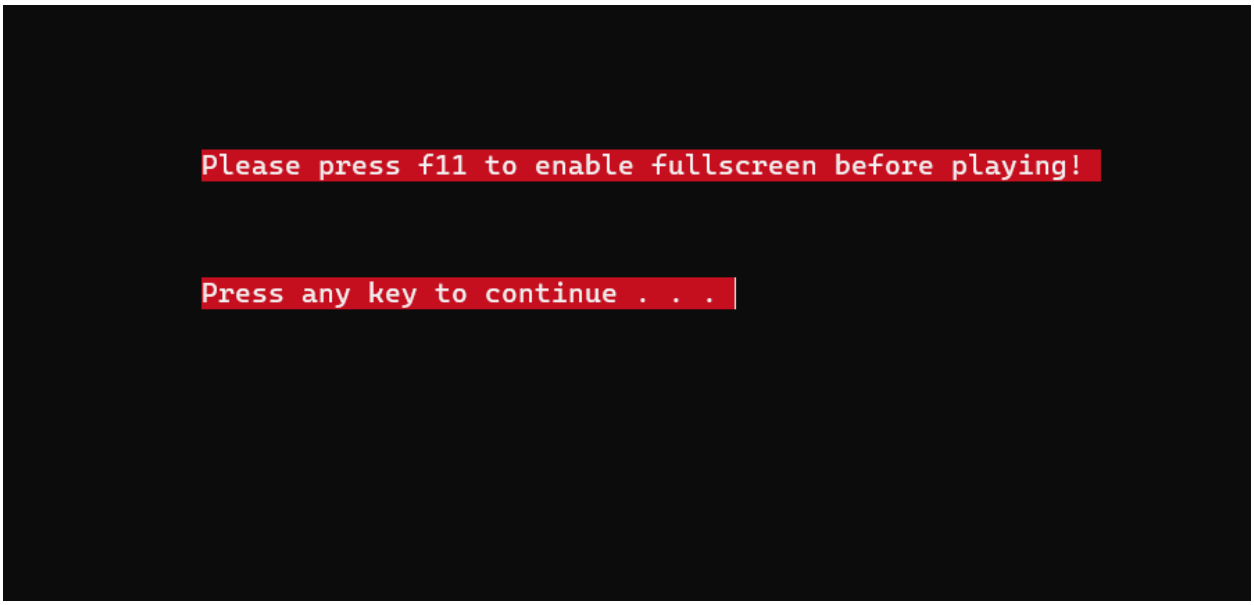

Figure 4 Notification to turn on full screen

After doing that, your game experience will be better and avoid unnecessary errors, and then press any key to continue.

2.4 Menu

This is the menu screen after switching to full-screen mode. Our game is named "Pikachu Matching." Here, our information is displayed, along with a table offering four different options, each corresponding to an application as indicated by its name. To get an idea before making a selection, you can refer to our Case Diagram.

You will use the **ARROW KEYS** on the keyboard to navigate between the options and press the **ENTER KEY** to make a selection.

We would like to express our gratitude to Lam Tien Huy (22127151) and Nguyen The Hien (22127107) for allowing us to reference their Pikachu drawing and the game title.

Figure 5 Menu

2.5 Help

This table displays the basic rules of the game to ensure the best experience. After the software presents the game board, you will navigate using the **ARROW KEYS** and make selections with the **ENTER KEY.**

To return to the menu, you can use the **BACKSPACE KEY**.

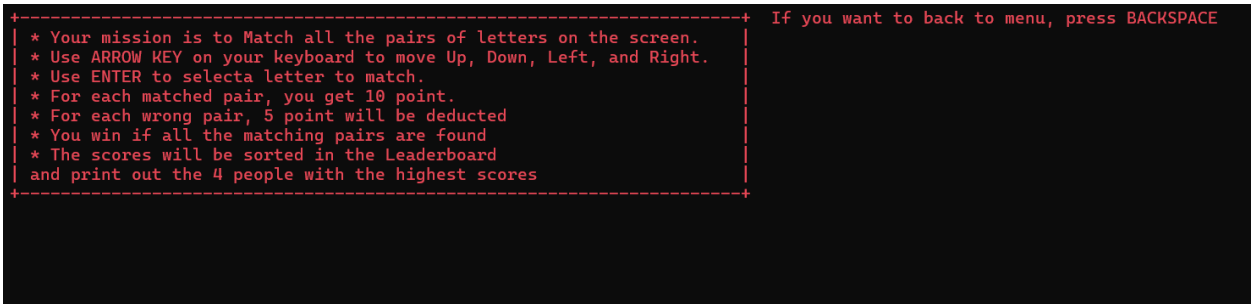

Figure 6 Help

2.6 Play

After logging into the game, users will choose the difficulty level they wish to play at. **Easy** will be the simplest mode, and **Hard** will be extremely challenging. The navigation and selection method remains the same as the Menu setup.

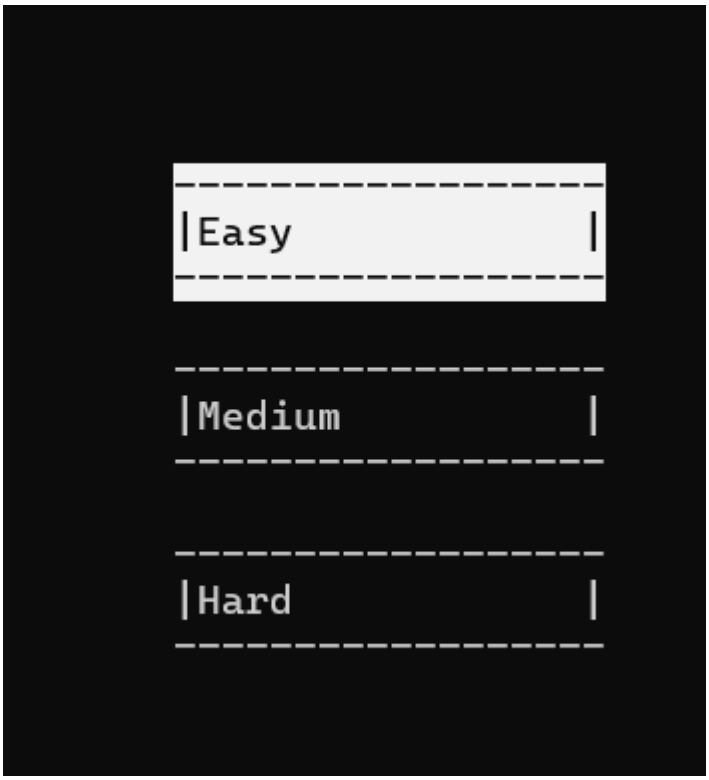

Figure 7 The table shows the level of difficulty

Next, players will proceed to the section to enter information for their Account. It will include the Player's Name along with the Student ID if applicable.

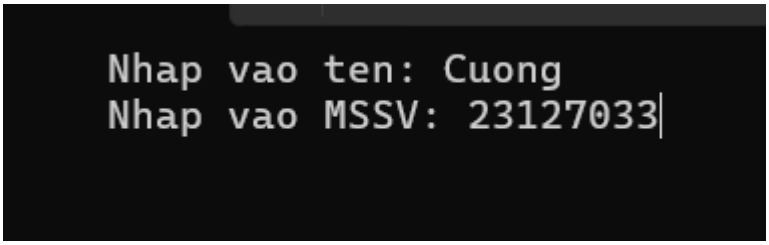

Figure 8 Account information

Following that is the main game screen, which will feature a board filled with characters that need to be connected. On the right side, there will be a summary panel including: the player information just entered; the score; and buttons for Move Suggestion, Help, Exit, which will be further explained in this report.

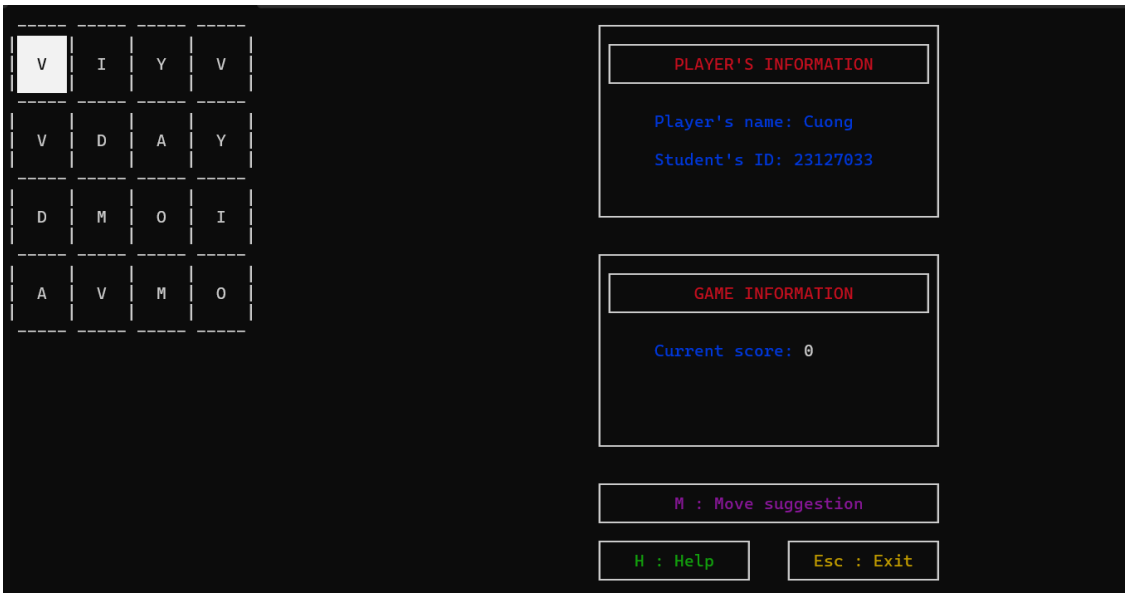

Figure 9 Main gameplay

We would like to extend our thanks to Tran Tung Lam (21127337) and Le Minh (21127165) for their assistance, which allowed us to reference the summary board section.

2.7 Result

If you complete the game, it will take you to a victory screen that includes the option to continue playing or to stop. If you choose Yes, you will proceed to play at the next level of difficulty; if you choose No, you will exit the game.

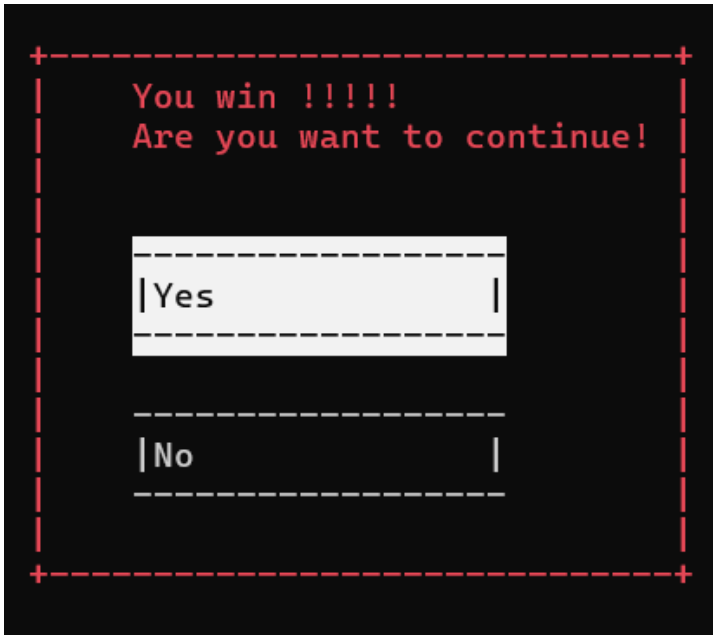

Figure 10 You Win!

If there are no more moves left to match and characters still remain on the board, you will lose. Please wait a moment, as the system will reset the board and allow you to start the game over.

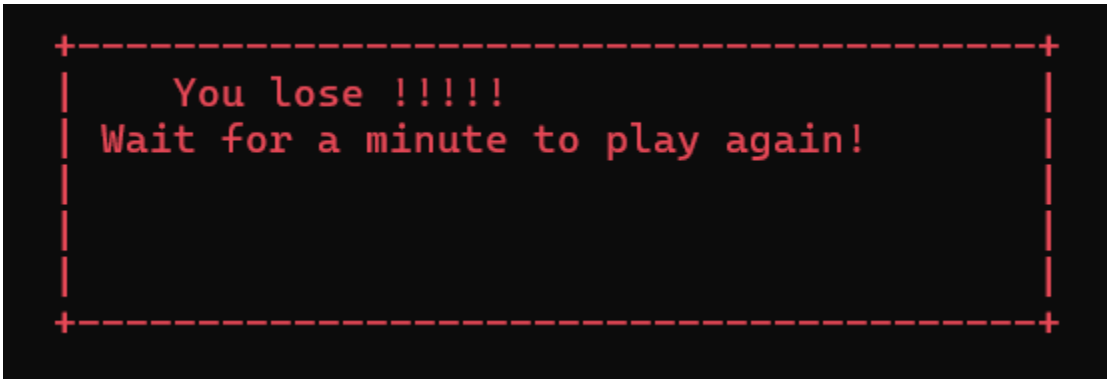

Figure 11 You Lose

Chapter 3

Progamming Explained

In this chapter, we will explain the libraries, header files, .cpp files, and how the game operates. Here, we use the C++ programming language.

Figure 12 C++ language (Wikipedia)

3.1 File Distribution

Our "The Matching Game" includes 1 Main.cpp, 5 .h files, and 5 .cpp files. Each file has a different setup with various functions.

- ***Main.cpp***

  This is the file that will run the previously installed programs and also display the necessary notifications.

Header files included: Menu.h, Controller.h

- ***Controller.h***

  This file compiles all the libraries needed for the program, accompanied by definitions of colors, arrow buttons, and sound files used in the program.
  Libraries inlcuded: isostream, string, window.h, vector, conio.h, cstdlib, fstream, algorithm, iomanip

- ***Controller.cpp***

  This file will handle the game control functions, such as moving the mouse cursor, playing music, adjusting console color, clearing the screen, and hiding or showing the cursor on the screen,….
  Header files included: Controller.h

- ***Image.h***

  This is the file that will manage the program's images and link it with Menu.h to enable printing images to the console.
  Header files included: Controller.h, Menu.h

- ***Image.cpp***

  This file will manage the printing of images on the screen, while the specific locations where they are printed will be handled in Menu.cpp.
  Header files included: Image.h

- ***Menu.h***

  This file defines the structure for storing player information, and defines functions to control the display of menu options on the screen, along with printing player information.
  Header files included: Controller.h, Image.h

- ***Menu.cpp***

This file displays the initial choices to the player and directs them to the applications associated with their selection. It is essentially regarded as the starting point of our entire game.
Header files included: Controller.h, Image.h, Menu.h, Playgame.h

- ***Matching.h***

  This file defines the functions that manage the game's logic. If the previous files represent the surface, this one is the depth of the game, where the methods of executing key logics are managed.
  Header files included: Menu.h, Controller.h, Playgame.h

- ***Matching.cpp***

  This is how the game's logics are configured, with functions to check patterns like U, I, L, or Z. It also includes Move Suggestion and checks whether pairs have been connected to display the appropriate colors.
  Header files included: Matching.h

- ***Playgame.h***

  This file will manage the display of game boards and backgrounds, and check whether the player wishes to continue.
  Header files included: Controller.h, Menu.h

- ***Playgame.cpp***

  This is the file that runs to draw the boards on the screen, and as pairs are connected according to specific logic, the background gradually reveals itself. This is likely one of the most crucial files in the program.
  Header files included: Playgame.h, Matching.h
  Libraries included: iomanip

- ***Leaderboard.txt***

  This file is used to store player information and also assists in printing out the leaderboard..

- ***Easy.txt***

Used to save easy mode background.

- ***Hard.txt***

  Used to save hard mode background.

- ***Medium.txt***

  Used to save medium mode background.

- ***All file .wav***

  Used to save sound.

3.2 Code Analysis

In Chapter 2 – Game Tutorial, we introduced the Case Diagram, and now we will explain how we set up the code to run the game as intended.

3.2.1 Libraries

- ***Iostream***

  It is a preprocessor directive in C++ that allows you to use input/output, the basics of C++.

- ***String***

  We use this library to store strings of characters for display on the screen as well as to save player information.

- ***Window.h***

  This library is used to configure settings for the console window, such as adjusting colors, hiding or showing the cursor, moving, and so on.

- ***Vector***

  We set up this library to store strings of characters in tables when displaying various options.

- ***Conio.h***

This library works in conjunction with Window.h to ensure that the program runs smoothly and without errors.

- ***Cstdlib***

  Declaring this library is for the purpose of generating characters randomly within a table..

- ***Fstream***

  This is the library used for reading and writing files; reading files will save the player's information..

- ***Algorithm***

  This library is set up to run comparison logic suitable for the game.

- ***Iomanip***

  This library is used to align margins and combine with others to draw tables with various options within an attractive frame.

3.2.2 Controller.h and Controller.cpp

In these code files, we are extremely grateful for the guidance from Lam Tien Huy (22127151), Nguyen The Hien (22127107), Tran Tung Lam (21127337), and Le Minh (21127165), along with the assistance of ChatGPT.

- ***Void playSound(int i);***

  Based on the definitions of the sound file names to be played, this function will play the audio files that we call during the operation of the game.
  Reference: Tran Tung Lam (21127337) and Le Minh (21127165) project. In file Controller.cpp, from line 128 to line 133.

- ***void gotoXY(int x, int y);***

This function will use the input values x and y to move the cursor on the console to that position to perform the operations that we have set up.
Reference: ChatGPT.

- ***void setConsoleColor(WORD color);***

  This function changes the color of the console at the desired position to the input color value.
  Reference: ChatGPT

- ***void clearScreen();***

  This is the function that will clear the screen.
  Reference: ChatGPT

- ***void TextColor();***

  This function also adjusts the text color in our game, similar to setConsoleColor; we have added this setting for easier recognition.
  Reference: Lam Tien Huy (22127151) and Nguyen The Hien (22127107) project. In file SetConsole.cpp, from line 109 to line 113

- ***void showCursor(bool show);***

  This function is used to show or hide the cursor on the console screen, which helps to enhance the game interface. Setting it to True will show the cursor, while setting it to False will hide it.
  Reference: Tran Tung Lam (21127337) and Le Minh (21127165) project. In file Controller.cpp, from line 54 to line 58.

3.2.3 Image.h and Image.cpp

- ***void displayLogo(int x, int y);***

  This function is used to draw the "Pikachu Matching" logo for our game. The values x and y specify the position where the logo will appear.

Reference: Lam Tien Huy (22127151) and Nguyen The Hien (22127107) project. In file SetConsole.cpp, from line 132 to line 148.

- ***void displayConrua(int x, int y);***

  This function is used to draw an image of a turtle-like Pokemon. The values x and y determine the position where the image will appear.
  Reference: Lam Tien Huy (22127151) and Nguyen The Hien (22127107) project. In file SetConsole.cpp, from line 266 to line 306.
- ***pikachuMenu(int x, int y);***

  This function is used to draw Pikachu on the menu screen. The values x and y specify the position where Pikachu will appear.
  Reference: Lam Tien Huy (22127151) and Nguyen The Hien (22127107) project. In file SetConsole.cpp, from line 248 to line 365.

3.2.4 Menu.h and Menu.cpp

In these code files, we developed the ideas ourselves and also utilized the support of ChatGPT to perfectly complete the segments of code. The struct Leaderboard L, which stores player information, is defined and frequently used across various functions.

- ***void drawMenu(const vector<string>& options, int highlightIndex);***

  This function is designed to draw boxes for the different option choices displayed on the board. The highlightIndex is the position of the currently selected option, and it changes the color of that box to white to make it easily recognizable for users.
- ***void menu(leaderboard L[100]);***

  This function is set up to print a menu that includes four choices: “Start”, “Leaderboard”, “Help”, and “Exit”. Each option has corresponding functionalities. For a more detailed understanding, you can refer back to the Case Diagram.

- ***void printRectangle(int left, int top, int width, int height);***

  This function is used to draw square or rectangular boxes based on the input values. The 'Left' and 'Top' parameters define the top-left position of the box, which is considered the starting point. 'Width' and 'Height' determine the width and height of the box that we set up.

- ***void printInterface(leaderboard L[100]);***

  This function is used to draw boxes, working in conjunction with the printRectangle function mentioned earlier. Inside this function, it processes information as well as determines the positions where the necessary boxes should be drawn.

- ***void stageDifficult(leaderboard L[100]);***

  This function is set up and configured to run when the user clicks "Start". It displays a board similar to the one in the menu, with boxes representing different difficulty levels for the rounds that players can choose.
  Additionally, this function also begins reading from and writing to a file to record player information.

- ***void printLeaderboard(string fileName, leaderboard L[100]);***

  This function is designed to display the leaderboard after a player finishes their game. It is set up to print the scores of the top four players on the screen.
  When a user clicks on the "Leaderboard" option in the menu, the leaderboard will be displayed.

- ***void printHelp(leaderboard L[100]);***

  This function is designed to display the game rules as well as basic information about the game's logic, including how points are added or

deducted, to best assist the player. This function will be executed when the player selects the "Help" option from the menu.

- ***void drawExit(const vector<string>& options, int highlightIndex);***

  This function is set up similarly to drawMenu. It draws options for the user to choose from and highlights the box when the cursor moves over it. This visual cue helps users easily identify and select their desired option.

- ***void printExit(leaderboard L[100]);***

  This function is activated when the player selects the "Exit" option. It displays a control panel with the choices "Yes" or "No". If "Yes" is selected, the program will terminate; if "No" is chosen, the program will return to the menu for further actions.

3.2.5 Matching.h and Matching.cpp

This function checks whether two cells can form a valid path. We handle this by conducting checks in two stages:

Stage 1: Checks if the two cells contain the same character and have different coordinates. If these conditions are met, it proceeds to Stage 2.

Stage 2: Determines if there is a valid path connecting the two cells. A path is considered valid if it forms the shape of an I, L, U, or Z (excluding the two cells themselves), and there are no obstacles blocking the path between them. This check is crucial for determining if a move is allowable according to the game's rules.

For convenition:

We will agree on the following conventions:

The x-coordinate is the column coordinate of a cell (counted from left to right).

The y-coordinate is the row coordinate of a cell (counted from top to bottom).

A cell with a '-' value on the board indicates that it is empty and can be traversed.

- ***bool checkLineX(int x1, int y1, int x2, int y2, char** board);***

  This function checks if there is a valid "I" shaped path (lying on a single row) between two cells (excluding the two cells themselves). We handle this by iterating from min(x1, x2) + 1 to max(x1, x2) - 1, where x1 and x2 are the column coordinates of the two cells. If any cell encountered during the iteration is not marked with '-', indicating an obstacle, the function returns false. If the path is clear of obstacles, it returns true. This ensures that the path is valid only if it is entirely unobstructed between the two specified cells.
- ***bool checkLineY(int x1, int y1, int x2, int y2, char** board);***

  Similarly, this function checks if there is a valid "I" shaped path (lying on a single column) between two cells (excluding the two cells themselves). We handle this by iterating from min(y1, y2) + 1 to max(y1, y2) - 1, where y1 and y2 are the row coordinates of the two cells. If any cell encountered during the iteration is not marked with '-', indicating an obstacle, the function returns false. If the path is clear of obstacles, it returns true. This ensures that the path is valid only if it is entirely unobstructed between the two specified cells vertically.
- ***bool checkL(int x1, int y1, int x2, int y2, char** board);***

  The purpose of this function is to check if there is a valid "L" shaped path between two cells (excluding the two cells themselves). The function processes through the following steps:

Operation Mechanism:

Step 1: If the two points are on the same row or column, return false, because an "L" shape cannot be formed if x1 = x2 or y1 = y2.

Step 2: For two cells that are neither on the same row nor column, identify two potential "L" bend points with coordinates (x1, y2) and (x2, y1). Then, check if these bend points are marked with '-'. If neither of the bend points is '-', (meaning there is no free path), return false. If one of the bend points is '-', proceed to Step 3.

Step 3:

Case 1: If (x1, y2) is '-', check for an "I" shaped path (lying on the same column) starting from (x1, y1) downwards to (x1, y2), and then check for an "I" shaped path (lying on the same row) from the bend point (x1, y2) to (x2, y2). If both paths are clear, return true; otherwise, return false.

Case 2: If (x2, y1) is '-', check for an "I" shaped path (lying on the same row) starting from (x1, y1) to (x2, y1), and then check for an "I" shaped path (lying on the same column) from the bend point (x2, y1) to (x2, y2).

- ***bool checkZ(int x1, int y1, int x2, int y2, char** board);***

  ĐThis function checks whether there is a valid 'Z' shaped path between two cells (excluding the two cells themselves). We distinguish two cases of a valid 'Z' shape: one where the path goes down from the first cell, moves across, and then continues down to the second cell, and another where it moves horizontally from the first cell, down or up, and then horizontally to the second cell. Brute force is the algorithm applied in this case.

Case 1: Down, across, then down.

- Step 1: Iterate from min(y1,y2)+1 to max(y1,y2)-1. For each y in the loop, check if the two bend points (x1,y) and (x2,y) are '-' (empty). If both are '-', proceed to step 2.

- Step 2: Check the path from (x1,y1) to (x1,y) using checkLineY and from (x1,y) to (x2,y) using checkLineX, then check from (x2,y) to (x2,y2) using checkLineY. If all conditions are met, return true (a 'Z' shaped path is found).

Case 2: Horizontal left, down or up , then horizontal.

- Step 1: Iterate from min(x1,x2)+1 to max(x1,x2)-1. For each x in the loop, check if the two bend points (x,y1) and (x,y2) are '-'. If both are '-', proceed to step 2.

- Step 2: Check the path from (x1,y1) to (x,y1) using checkLineX and from (x,y1) to (x,y2) using checkLineY. Finally, check if the path from (x,y2) to (x2,y2) is valid. If all conditions are met, return true (a 'Z' shaped path is found).

- ***bool checkU(int x1, int y1, int x2, int y2, char** board, int m, int n);***

  Purpose: The checkU function is used to determine if there is a valid 'U' shaped path between two cells on the board, excluding the starting and ending cells. A 'U' can be formed in four different orientations: two horizontal (with the vertical segment either on the right or left) and two vertical (with the horizontal segment either at the top or bottom).

Operating Mechanism: The function employs a brute force technique to check each possibility of forming a 'U' shape through loops and auxiliary functions (checkLineX and checkLineY) to validate straight segments.

Case 1: Horizontal U with the vertical segment on the right

- Step 1: This step verifies if a 'U' can be formed outside the board's boundaries. It checks whether a horizontal path can be drawn from (x1, y1) to the end of the row and from (x2, y2) to the end of the row. If both are feasible, return true.

- Step 2: Iterate from max(x1, x2) + 1 to n-1 to find a valid position i where both bending points board[y1][i] and board[y2][i] are marked as '-'.

- Step 3: If found, verify the horizontal path from (x1, y1) to (i, y1), the vertical path from (i, y1) to (i, y2), and finally the horizontal path from (i, y2) to (x2, y2). If all checks are validated, return true.

Case 2: Horizontal U with the vertical segment on the left

- Step 1: This step checks if a 'U' can be created outside the board's boundaries by checking if a horizontal path from (x1, y1) to the start of the row and from (x2, y2) to the start of the row is feasible. If both conditions are met, return true.

- Step 2 and Step 3: Similar to Case 1, but start checking from min(x1, x2) - 1 and move towards 0.

Case 3: Vertical U with the horizontal segment below

- Step 1: This step checks if a 'U' can be created outside the board's boundaries by checking if a vertical path from (x1, y1) down to the end

of the column and from (x2, y2) also to the end of the column is feasible. If both conditions are met, return true.

- Step 2: Iterate from max(y1, y2) + 1 to m-1 to find a valid position i where both bending points board[i][x1] and board[i][x2] are marked as '-'.

- Step 3: Verify the vertical path from (x1, y1) to (x1, i), the horizontal path from (x1, i) to (x2, i), and finally the vertical path from (x2, i) to (x2, y2). If all steps are validated, return true.

Case 4: Vertical U with the horizontal segment above

- Step 1: This step checks if a 'U' can be created outside the board's boundaries by checking if a vertical path from (x1, y1) up to the start of the column and from (x2, y2) also to the start of the column is feasible. If both conditions are met, return true.

- Step 2 and Step 3: Similar to Case 3, but start checking from min(y1, y2) - 1 and move towards 0.

- ***bool isFinished(char** board, int m, int n);***

  The game ends if one of the following two conditions occurs:

  All cells have the value '-'.

  There are no more valid moves left.

  Condition 1:

  The function bool isFinished returns true if all cells have the value '-'; it returns false if there is at least one cell with an alphabetic value.

  Operation Mechanism: The function iterates through the entire game board. If it finds a cell with a character other than '-', it returns false.

If it completes the iteration without finding any such cell, it returns true, indicating that the game has ended.

Condition 2:

The function MoveSuggestion is designed to check if there are any valid moves left on the current board according to specific rules. It returns true if it finds at least one valid move between two cells with the same value but different positions.

Operation Mechanism:

Step 1: Traverse the board. The function starts by iterating through each cell on the board using two nested loops, with x1 and y1 being the indices of the current cell.

Step 2: Check non-empty cells. It only considers cells that are not empty ('-').

Step 3: Search for a matching pair. It continues to iterate through the board to find a second cell, indexed by x2 and y2, such that this cell is in a different position from the first cell and has the same value as the first cell.

Step 4: Validate the move. It uses a series of functions (checkU, checkLineX, checkLineY, checkZ, and checkL) to determine if there is a valid path from the first to the second cell. These functions check for different types of paths such as U-shaped, Z-shaped, horizontal lines, vertical lines, and L-shaped.

Step 5: Return the result. If any path-checking function returns true, MoveSuggestion will also return true; otherwise, if no valid paths are found, the function returns false.

- ***void checkeasy(char**& board, int x1, int y1, int x2, int y2, int m, int n, int& score);***

  This function serves to execute the above checking functions specifically for the easy mode of the game. Here's how it works:
  If the check is true: The function will highlight the cell in blue, indicating a correct match, and then the cell will disappear as part of the matching process. This indicates to the player that they have successfully found a matching pair.
  If the check is false: The function will highlight the cell in red, signaling an incorrect match or move. The cell will remain visible, allowing the player to try again or choose another move. This feedback helps players learn from their mistakes and understand the game better without penalizing them immediately for incorrect attempts.
- ***void checkmedium(char**& board, int x1, int y1, int x2, int y2, int m, int n, int& score);***

  Similar to the easy mode, this function is used for the medium mode of the game, with the following specifics:
  If the check is true: The function highlights the cell in green, indicating a correct match, and then makes the cell disappear. This successful action confirms to the player that they have found a correct pair, which is then cleared from the board to reflect their progress.
  If the check is false: The function highlights the cell in red, indicating an incorrect match. The cell remains visible, allowing the player to reassess their strategy and try again. This feedback is crucial as it helps maintain the challenge in the medium mode, encouraging players to think more carefully about their moves without immediate severe penalties.
- ***void checkhard(char**& board, int x1, int y1, int x2, int y2, int m, int n, int& score);***

Just like checkEasy and checkMedium, this function is tailored for the hard mode of the game.

- ***bool MoveSuggestion(char** board, int m, int n);***

  The function you described helps players determine the next valid move by identifying two cells that can be connected according to the game's rules. Here's how the function operates using a brute force algorithm:

  Operation Mechanism of the Function:

  Step 1: Loop through the board: The function starts with two nested loops (using x1 and y1) to iterate through each cell on the board.

  Step 2: Check the condition of the cell: For each cell at position (x1, y1), the function checks if it is not a '-' character. If it is not '-', the function proceeds with the following steps.

  Step 3: Find a corresponding pair: The function uses two additional loops (using x2 and y2) to find another cell on the board that has the same value as the cell at (x1, y1). It ensures that (x2, y2) is not the same as (x1, y1).

  Step 4: Check connectivity: If two cells have the same value, the function checks if they can be connected through one of the path-checking functions such as checkU, checkLineX, checkLineY, checkZ, or checkL.

  Step 5: Highlight possible moves: If one of the path-checking functions returns true, the cells will be highlighted (using the highlightCellforEnter function), the program will pause briefly (e.g., Sleep(500)), then the highlight is removed, and the function returns true.

  End: If no pairs meet the criteria, the function concludes and returns false.

- ***bool MoveSuggestionUpdate(char** board, int m, int n);***

This function uses a similar logic to the one previously described but includes an additional feature to enhance gameplay: it highlights the cells that meet the conditions in blue. This visual aid makes it easier for players to spot valid moves, thereby improving the gaming experience.

3.2.6 Playgame.h and Playgame.cpp

In these files, the ideas come from us, with the support of ChatGPT 4, to complete an important part of the game: the game board with various characters.

- ***void drawBoard2P(char** board, int rows, int cols);***

  This function operates by drawing a grid with rows and columns whose values are specified. The grid will position character letters in the center, and each cell will have a width of 7 and a height of 5. Around each cell, there will be additional characters "|" and "-" to clearly delineate the cells.

- ***void createBackgroundE(string* background, int& i);***

  This function is used to read the background characters from the file "easy.txt" and load them into an array of string pointers, which allows access to each row and column. It can be considered as storing the image in a two-dimensional array to preserve the size and position of each character.

- ***void printBackGroundE(int x, int y);***

  The idea is to overwrite the characters stored in the array of string pointers onto the original game board when a cell disappears.
  This function is used to draw the background when a cell has disappeared, with x and y specifying the location of that cell. Since each cell additionally includes width and height dimensions, it may not align perfectly with the image saved in the createBackgroundE

function, necessitating the use of several conditional statements (if statements) to ensure that the characters are printed in positions on the board that match the locations of the characters stored in the array.

- ***void printBackGroundHighLightE(int x, int y);***

  This function is a direct copy of the printGroundE function, but with a modification to the coordinates of the cells from left to right, which are incremented and decremented by one. This adjustment is because when highlighting, only the inner part of the cell is highlighted, not the entire surrounding border. Meanwhile, for drawing purposes, it's necessary to overwrite including the borders around the cells. This distinction ensures that the visual representation within the game accurately reflects both highlighted states and the static background structure.

- ***void createBackgroundM(string* background, int& i);***

  Similar to createBackgroundE, this function reads from the file "medium.txt".

- ***void printBackGroundM(string* background, int x, int y);***

  Similar to printBackGroundE, but this time since the values for x and y are greater, additional if statements are needed to fully render the background.

- ***void printBackGroundHighLightM(string* background, int x, int y);***

  This function has the same concept as printBackGroundHighLightE.

- ***void createBackgroundH(string* background, int& i);***

  Similar to the functions for Easy and Medium, but this time it reads from the file "hard.txt".

- ***void printBackGroundH(string* background, int x, int y);***

Similar to printBackGroundE, but this time, since the values for x and y are greater, a few more if statements are needed to fully draw the background.

- ***void printBackGroundHighLightH(string* background, int x, int y);***

  Similar to printBackGroundHighLightM.

- ***void highlightCell(int x, int y, char** board, int cellWidth, int cellHeight, int cellPadding, bool highlight, string difficult);***

  This function highlights cells when navigating through them, meaning that the cell where the cursor is currently located will be highlighted in white. Additional conditions are included for each mode, as each mode features a background with different dimensions.
  When bool highlight is true, the function will perform the drawing with highlighting; otherwise, it will draw with a normal black background and white text.

- ***void highlightCellforEnter(int x, int y, char** board, int cellWidth, int cellHeight, int cellPadding, bool highlight, string difficult);***

  Similar to the previous function, this function also highlights a cell when the cursor enters it, but this time, the highlight will not be white; instead, it will turn blue, indicating that the user has selected that cell. The cell selected in blue will remain so until bool highlight is false, at which point it will revert to the normal black background with white text. This setup is designed for the Enter key functionality in our game.

- ***void highlightCellforEnterError(int x, int y, char** board, int cellWidth, int cellHeight, int cellPadding, bool highlight, string difficult);***

Similar to the ***highlightCellforEnter*** function, this function changes the highlight color to red instead of blue. We have set this up to indicate incorrect selections by the user.

- ***void fillBoardWithPairs(char**& board, int m, int n);***

  First, we dynamically allocate the board with predefined rows and columns (4x4, 6x6, 8x8). Next, we follow steps to randomly assign char values from the alphabet (A, B, C, ..., Z) to the board, ensuring each character always appears in pairs.

  Step 1: The function first checks if the total number of cells is even because an odd total would result in one character appearing an odd number of times, which would be invalid.

  Step 2: Then, it creates an additional array, characters, with a size of m x n. Each pair of adjacent elements in this array is assigned a randomly selected alphabet character, ensuring that each character appears an even number of times.

  Step 3: The function then uses the random_shuffle method to randomly mix the characters in the characters array. Each element of the board is then assigned a value from the characters array, populating the board with evenly distributed, paired characters in a random arrangement.

- ***void drawContinue(const vector<string>& options, int highlightIndex);***

  This function is similar to the drawMenu or drawExit functions as it also draws a board with various options.

- ***void printContinue(bool& running,leaderboard L[100], string df);***

This function runs when the player has completed their current level and asks if they want to continue at a higher difficulty. If the player chooses "Yes", they will proceed to a harder level than the one initially selected. If they choose "No", the program will end.

- ***void easy(char** board, int m, int n, int& score,leaderboard L[100]);***

  This function is the main function that integrates the above functions—from drawing the background and game board to presenting a summary table. It includes control features using arrow keys on the keyboard for navigation and other controls like Enter or M for move suggestions. When it is determined that no more moves are possible, there are two scenarios:

  Case 1: All cells have been connected. In this case, the program will display the "Continue" board, asking the player if they want to proceed to the next level or challenge.

  Case 2: Some characters can no longer be connected. Here, the program will notify the player that they have lost and will pause briefly before offering the option to restart the game from the beginning.

- ***void medium(char** board, int m, int n, int& score, leaderboard L[100]);***

  Similar to easy function, but now the columns and rows are 6x6.

- ***void hard(char** board, int rows, int cols, int& score, leaderboard L[100]);***

- Similar to easy function, but now the columns and rows are 8x8.

3.3 Advanced Features

3.3.1 Color Effect and Sound Effect

We will use the color functions in Controller.h to highlight various parts as needed. For example, highlighting a cell when the cursor moves to it.

Figure 13 Color effect 1

Or highlight cells in the game board to make them more visible to users, providing a comfortable gaming experience. When cells are connected, additional colors will also be displayed.

Figure 14 Color effect 2

For Sound Effects, we use the playSound function set up in Controller.h to play music. For example, when a cell is selected, the command ***playSound(ENTER_SOUND);*** will be executed.

3.3.2 Visual Effect

We have created a game featuring a variety of colors and integrated images, making it vibrant and engaging for players.

Figure 15 Visual effect 1

Additionally, there are images displaying player information and continuously updated scores. When cells are connected, additional color effects and disappearing effects are also applied.

Figure 16 Visual effect 2

3.3.3 Background

As explained in section 3.2.6, when cells are successfully connected, they disappear, gradually revealing the background, making the game more lively. Below is an image of a background in easy mode that is about to be fully revealed.

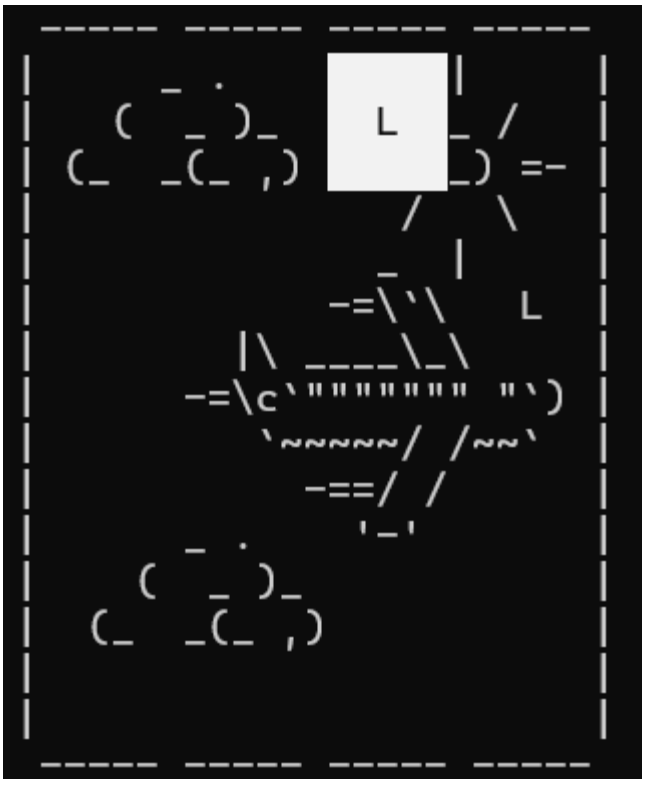

Figure 17 Background

3.3.4 Leaderboard

This feature displays the leaderboard of the players with the highest scores. We showcase the top four scorers who have played the game. The player history is preserved through the leaderboard.txt file, which is written in append mode. To display the leaderboard content, we reopen the same leaderboard.txt file to read and organize the contents using a leaderboard struct to be displayed on the console screen. Here are the detailed steps involved:

Step 1: Determine the user's menu selection.

Case 1:

If the user chooses to view the leaderboard, the leaderboard.txt file is opened, and a while loop is used to sequentially read four pieces of data: Name, MSSV (student ID), Score, and Difficulty from the file.

Sort the data using the Interchange Sort algorithm to arrange the leaderboard struct in descending order of scores.

Display the data: run a loop for(i=0; i<min(t,4); i++) where t is the number of data entries saved in the leaderboard.txt file. This loop displays exactly the top four highest scorers.

Case 2:

If the user chooses to start and selects a difficulty level (easy, medium, hard), they are prompted to enter their name and student ID.

After entering this information, the leaderboard.txt file is opened in append mode to save the player's profile..

3.3.5 Move Suggestion

As mentioned in section 3.2.5, the functions operate by displaying a pair of characters that can be connected on the screen, highlighting those cells in green, and then reverting them back to normal. This aids users by visually indicating potential moves, making the gameplay more accessible and user-friendly. This feature not only enhances the interactive aspect of the game but also helps players identify possible actions they might not have noticed, thus improving their gaming strategy.

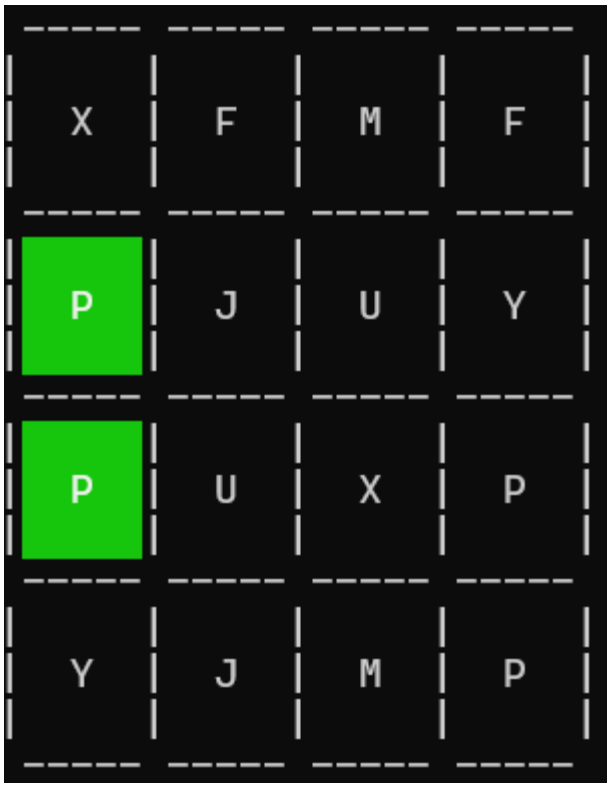

Figure 18 Move suggestion

3.3.6 Game Account

As described in section 3.2.4, when players enter their information, it is saved in a file named Leaderboard.txt. At the end of the game, the player's final score is also recorded in this file along with the difficulty level they selected. This system ensures that all relevant player achievements are documented, allowing for easy tracking and comparison of scores across different game sessions and difficulty levels.

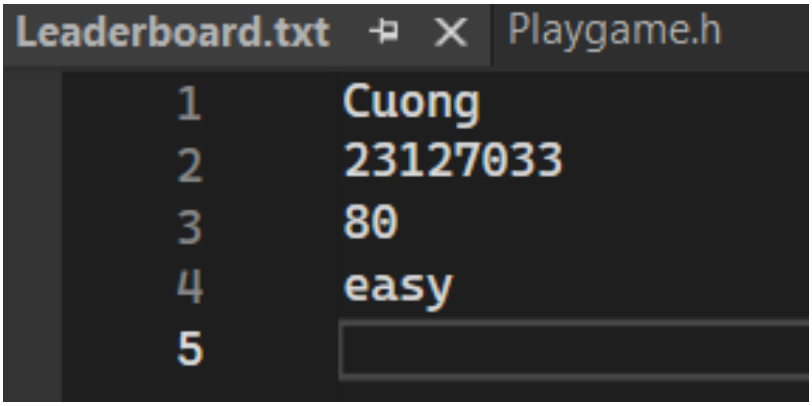

Figure 19 Game account save

Chapter 4
Uncoded Research

This section primarily explores our research, focusing on components of the project that were beyond our technical capabilities to implement. While we regret our inability to integrate these specific ideas into our programming, we believe including a detailed discussion of our research will enhance your understanding of our overall concept.

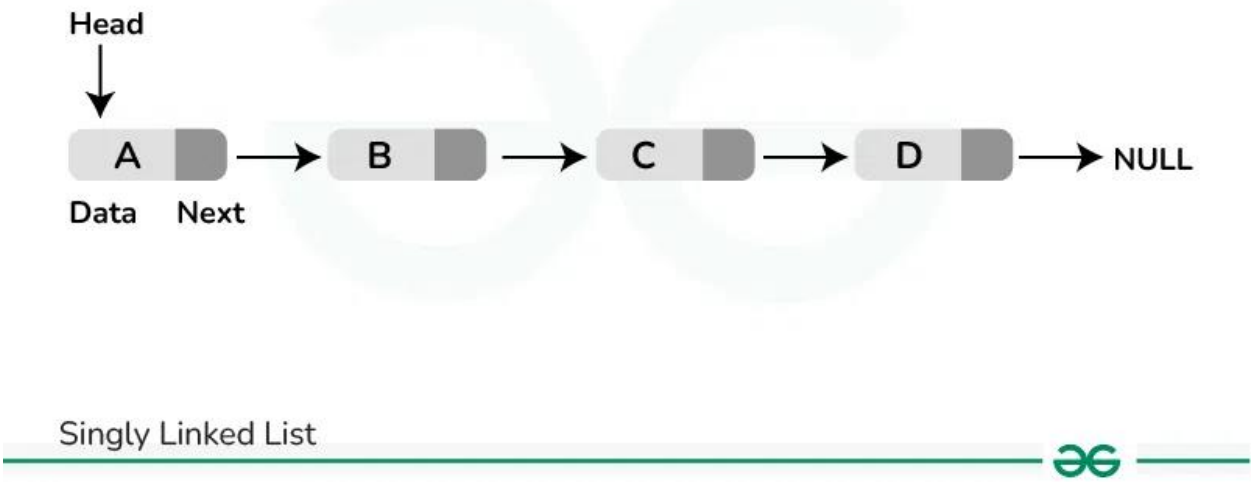

Figure 20 Linked list

Please note that these elements were not incorporated into our game; they represent our thought process and intentions, enriching the content of our project report.

4.1 Stage diffient increase

We are requested to create a more difficult stage after players finish the previous one by sliding the neighboring cells (from the game stage) into the newly emptied spaces in a particulardirection (left to right, up to down, etc.) after a matched.

4.1.1 Programming Direction

Step 1: identify the positions of the two removed cells on the board. Suppose you have their coordinates as (x1, y1) and (x2, y2).

Step 2: Write a function to slide cells

The sliding of cells depends on the direction you want to slide, such as from top to bottom. Here's a description of the function that slides cells from the top down to fill the empty spaces:

For each column containing a removed cell, start at the removed cell's position and check the cells above it.

Slide down: If a non-empty cell is found above, move its value down to the empty cell's position and mark that cell as empty. Continue this process until all the bottom cells in the column are filled or there are no more cells to slide.

Repeat until no more cells can be slid: This process needs to be repeated until there are no cells above to slide down, or the cells are already in the lowest possible position.

Step 3: Update the board after sliding

After the cells have been slid down, the board needs to be updated to reflect the changes in structure. You may need to redraw or refresh the display on the user interface.

Pseudo code for the slideDown function:

```
function slideDown(columnIndex, startRow)
    while startRow < number of rows in the board
        if cell at (startRow, columnIndex) is empty
            find k from startRow up until a non-empty cell is found or end of the board
            if a non-empty cell is found
                assign the value of the non-empty cell to the empty cell
                set the cell that was assigned to empty
            else
                break from the loop
        increment startRow by 1
end function
```

Data structures to implement this: 2D-Array Pointer and Linked List

2D-Array Pointer is the data structure we have used to code this project. In this case, you need to traverse to see which cell is empty on the board and then write a sliding function as described above to remove the cell and slide the cell above the removed cell down.

Linked List is a data structure where data is stored as a chain of nodes, with the last node pointing to NULL, and nodes are linked via the address of the next node. A doubly linked list node has three parts: data, the address of the previous node, and the address of the next node. A singly linked list node has two parts: data and the address of the next node.

In this case, we would use a linked list to store cell values. Each node would store the value of a cell. Deleting a cell is straightforward. We just need to write a delete Node function that connects the preceding node to the

node following the node to be deleted. To traverse nodes, you need a loop from the head node to the empty spot, delete the value inside the node, and assign it the value from the node in the row above..

4.1.2 Pointer and Linked List Comparison

<table>
<tr><td></td><td>Pointers</td><td>Linked List</td></tr>
<tr><td>Similarities</td><td colspan="2">Both operations are dynamically allocated on the heap. Both are data structures used for storage.</td></tr>
<tr><td rowspan="2">Data access</td><td>Random access value. Accessing the value is performed in constant time O(1).</td><td>Must traverse from the head to the element you want to find. In the worst case, you have to traverse all nodes. The average complexity is O(n).</td></tr>
<tr><td colspan="2">Pointers prove to be superior to Linked Lists if the algorithm needs to access elements frequently.</td></tr>
<tr><td rowspan="2">Matching</td><td>Pointers use individual functions for matching (U, I, L, Z).</td><td>Linked List uses one function for all matching tasks using the BFS (Breadth-First Search) algorithm.</td></tr>
<tr><td colspan="2">Linked List is more efficient in matching.</td></tr>
</table>

References

Special thanks to people who contributed to our project.

- Lam Tien Huy (22127151) and Nguyen The Hien (22127107): Visual effect and Reference sample for reports.
- Tran Tung Lam (21127337) and Le Minh (21127165): Visual effect and Controller.

- Louis2602 | Github | Pikachu – Game (April 22 2022): GitHub - Louis2602/Pikachu-Game: Pikachu Game made with C++ | HCMUS Programming Techniques Project
- Chat GPT 4: ChatGPT (openai.com)
  - GPT-4 suggests using the random_shuffle function to randomize values in the characters array in the void fillBoardWithPairs function.
  - GPT-4 suggests finding two bend points in the checkL function, significantly reducing the number of if loops compared to the original approach. This suggestion makes the code much easier to understand..
  - GPT-4 suggests using a separate highlightCell function each time a move is made. Initially, we had to clear the screen and redraw the entire board every time the user moved using the arrow keys, which caused flickering and lag. However, the suggestion to only redraw the cell that is moved to or selected upon pressing enter significantly improves the game's smoothness and user experience.
  - GPT-4 proposes functions for rendering choice boards in the Menu section and other areas requiring boards with multiple options. This facilitates a more structured and efficient way to handle user interactions across different parts of the game or application interface..