VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

MATCHING GAME

Topic: Console Game

Course: Programming Techniques

Students execute:
Tran Ly Nhat Hao - 23127187
Huynh Hao Nam - 23127431

Instructors provide guidance:
MSc. Bui Huy Thong
MSc. Nguyen Tran Duy Minh

227 Nguyen Van Cu, Ward 4, District 5, Ho Chi Minh city

10th April 2024

Contents

Student Information

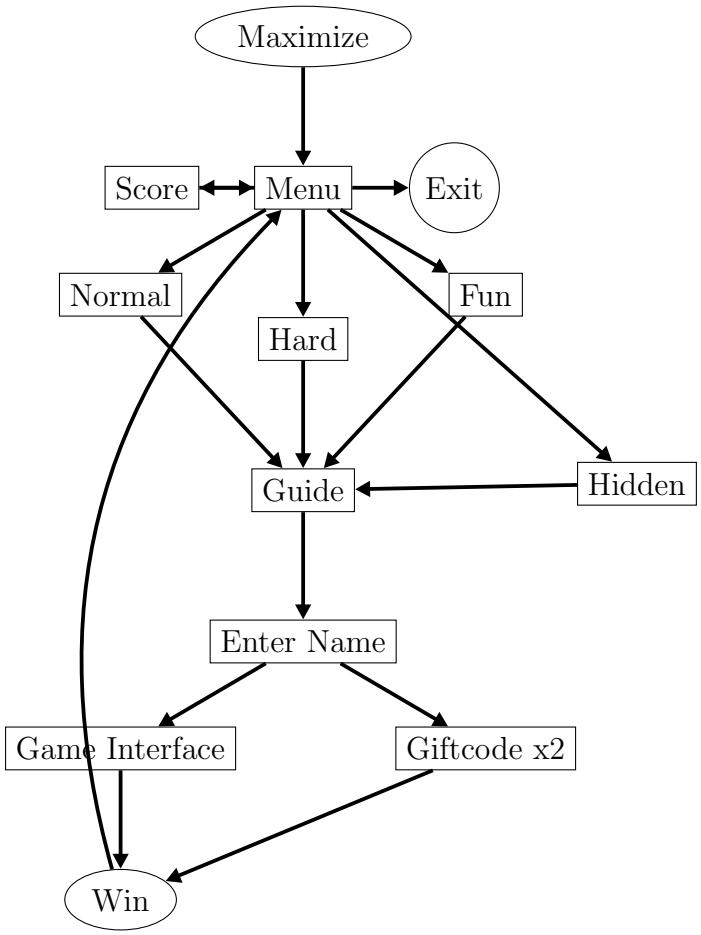
1. **Name:** Tran Ly Nhat Hao - 23CLC08
    - **Student ID:** 23127187
    - **Github:** tranlynhathao
    - **Course:** CSC10002 – ADVANCED PROGRAMMING TECHNIQUES AND PRACTICES

2. **Name:** Huynh Hao Nam - 23CLC08
    - **Student ID:** 23127431
    - **Github:** Hownameee
    - **Course:** CSC10002 – ADVANCED PROGRAMMING TECHNIQUES AND PRACTICES

3. **Github Repository:** pikachu_matching_game

```
.
├── LICENSE
├── Music
│   ├── Legend.wav
│   ├── buzzer.wav
│   ├── choose.wav
│   ├── mario.wav
│   └── sen.wav
├── My Matching Game.exe
├── README.md
├── Source
│   ├── Array.cpp
│   ├── Array.h
│   ├── Check.cpp
│   ├── Check.h
│   ├── Draw board and pokemon.cpp
│   ├── Draw board and pokemon.h
│   ├── Game Mode.cpp
│   ├── Game Mode.h
│   ├── Lib.h
│   ├── Main.cpp
│   ├── Move Suggest and Delete.cpp
│   ├── Move Suggest and Delete.h
│   ├── ReadFile.cpp
│   ├── ReadFile.h
│   ├── Redo.cpp
│   ├── Redo.h
│   ├── Struct.h
│   ├── Undo.cpp
│   └── Undo.h
└── Text
    ├── Hard.txt
    ├── fungame.txt
    ├── hidden.txt
    └── normal.txt

4 directories, 31 files
```



1 Introduction

The Matching Game (commonly known as Pikachu Puzzle Game) includes a board of multiple cells, each of which presents a figure. The player finds and matches a pair of cells that contain the same figure and connect each other in some particular pattern. A legal match will make the two cells disappear. The game ends when all matching pairs are found. Figure 1 shows some snapshots from the Pikachu Puzzle Game.

In this project, we will develop a simplified version of this Matching Game by remaking the game with characters (instead of figures).

Figure 1.1: The Pikachu Puzzle Game [2]

2 How the game works

2.1 Directory tree and Interface diagram

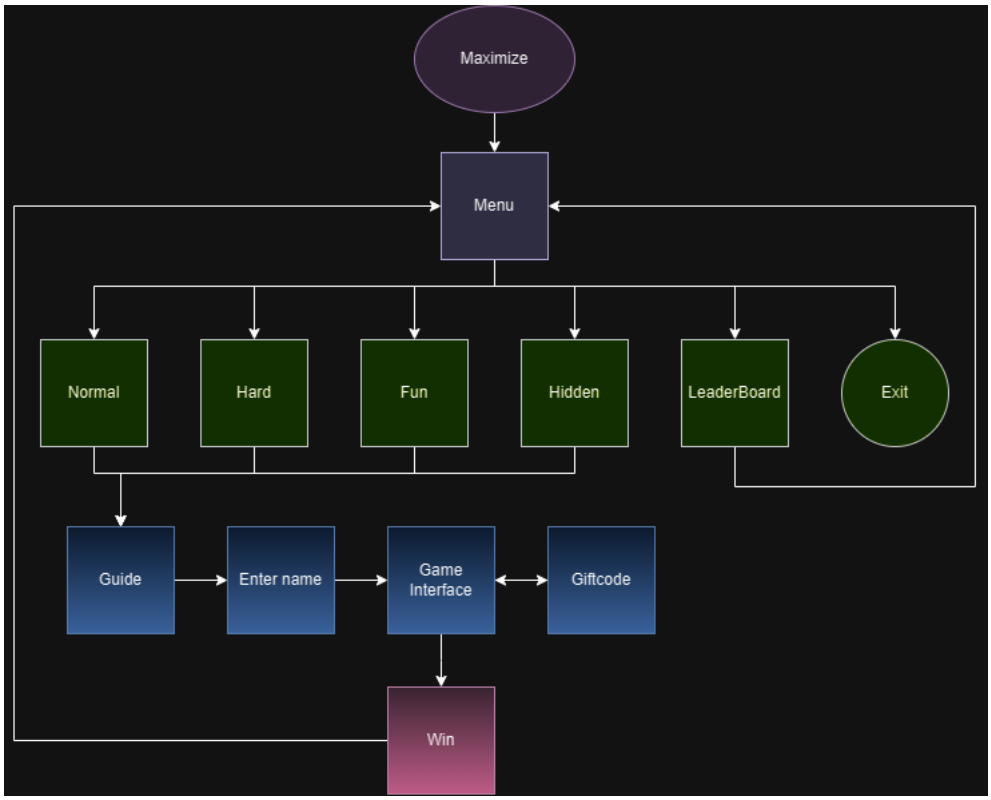


Figure 2.1: Use case diagram of Interface game

- Shortcut keys for guidance and keys to continue displaying the game will be available on each screen starting from the first screen.

2.2 Game Toturial

2.2.1 Enlarge the console screen:

♫ When the game starts, a violin music named "Senbonzakura" will be played until the main matching game interface officially begins. [3]

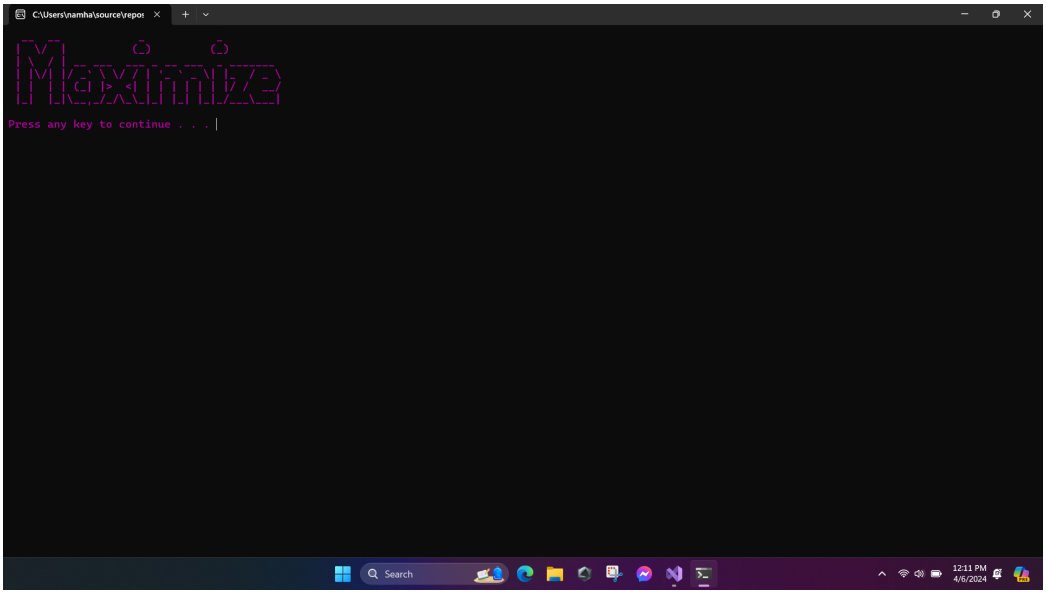


Figure 2.2: Maximize

2.2.2 Game menu

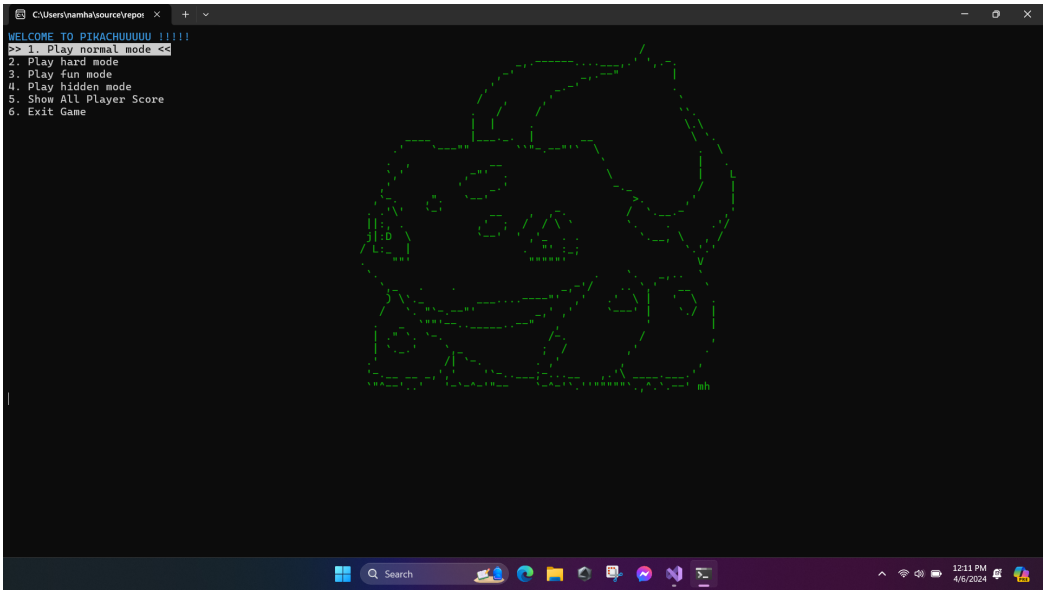


Figure 2.3: Menu game [4]

At the game menu screen, there will be displayed 6 options, each option pointed to by the cursor will show different interface:

- 1-4 are different game modes.
- 5 is to display the leaderboard or ranking board.
- 6 is the option to exit the game.

2.2.3 Control guide and gameplay

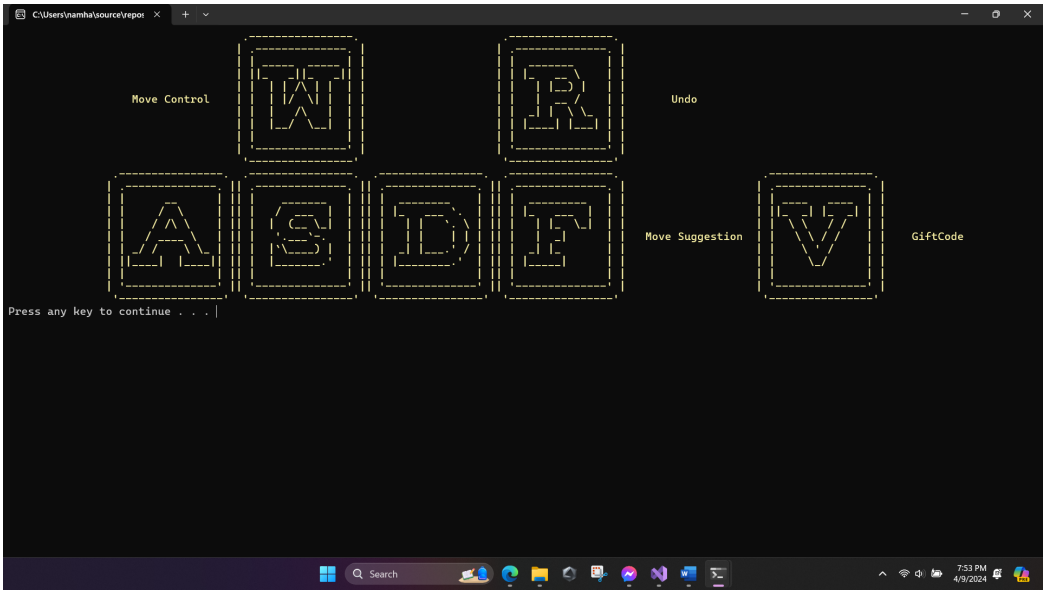


Figure 2.4: Control guide

Shortcuts:

- Use '**w**', '**a**', '**s**', '**d**' keys for movement, or use the navigator or enter row-column coordinates to move directly to that point.
- Press '**r**' to go back (undo).
- Press '**f**' for the next move hint.
- Press '**v**' to enter gift code.

2.2.4 Enter name

⚠ Note that the name MUST NOT CONTAIN SPACE, but it can include diacritics.

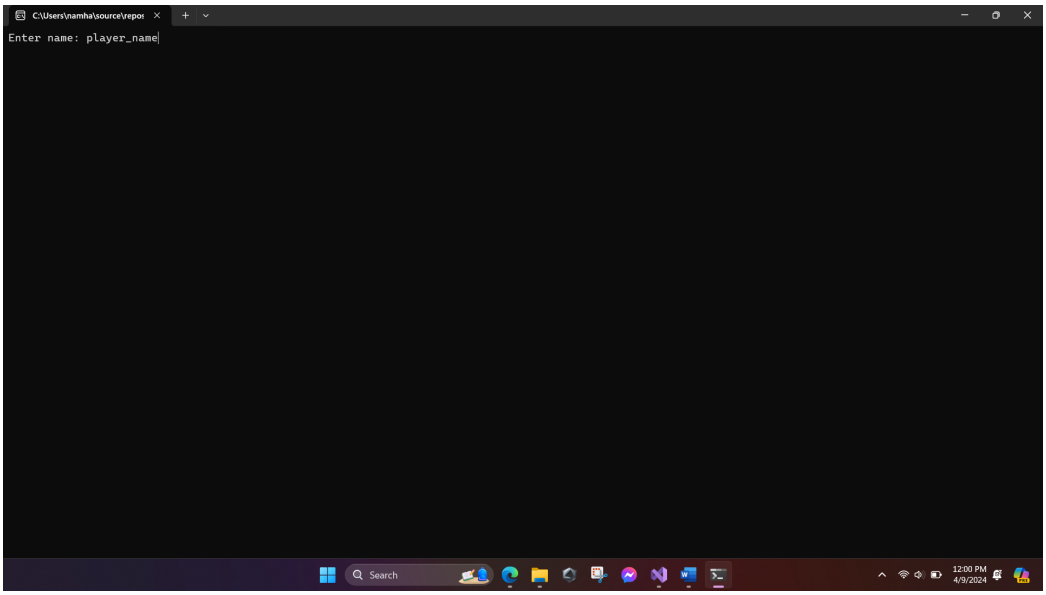


Figure 2.5: Enter name

2.2.5 Game screen

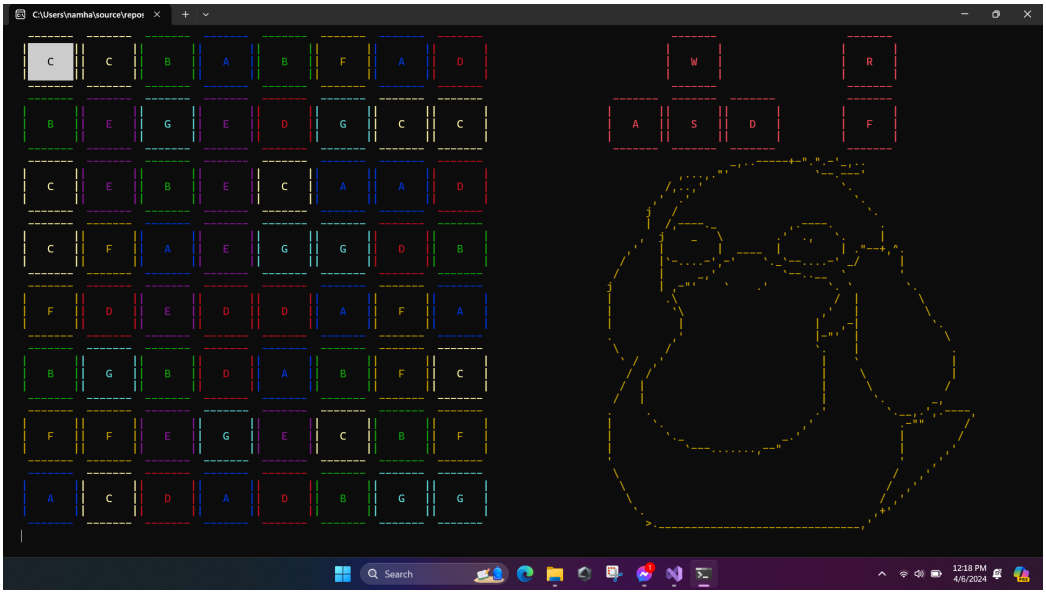


Figure 2.6: Interface of normal, hard, fun mode [4]

Rules:

- Players select 2 tiles with matching letters to remove, and there must be a path between the two letters. The number of connections should not exceed 3; otherwise, they cannot be connected. When there are no more tiles on the screen, the player wins.
- In Normal mode, the game operates normally without any changes.
- In Hard mode, when successfully removing any two tiles, the game will automatically move the last tile up to fill the space just cleared until all spaces are at the back and there are no spaces between the tiles. The rows also operate the same way.
  - SPECIAL: If you make one wrong selection, you will have to start over, meaning the screen will reset to the beginning of the game, so be careful, and you cannot UNDO if you get into this situation.
- In Fun mode, every time you select 2 tiles, whether right or wrong, the game will still randomly generate a new array and arrange the spaces like in Hard mode.

♫ Here, there will be sounds played when navigating and matching tiles in the game. [5] [6] [7]

2.2.6 "Special" mode (Hidden mode)

- **Hidden mode:** In this mode, all tiles are hidden and only appear when you select them. You have to find 2 matching tiles to connect if possible. Keep trying!

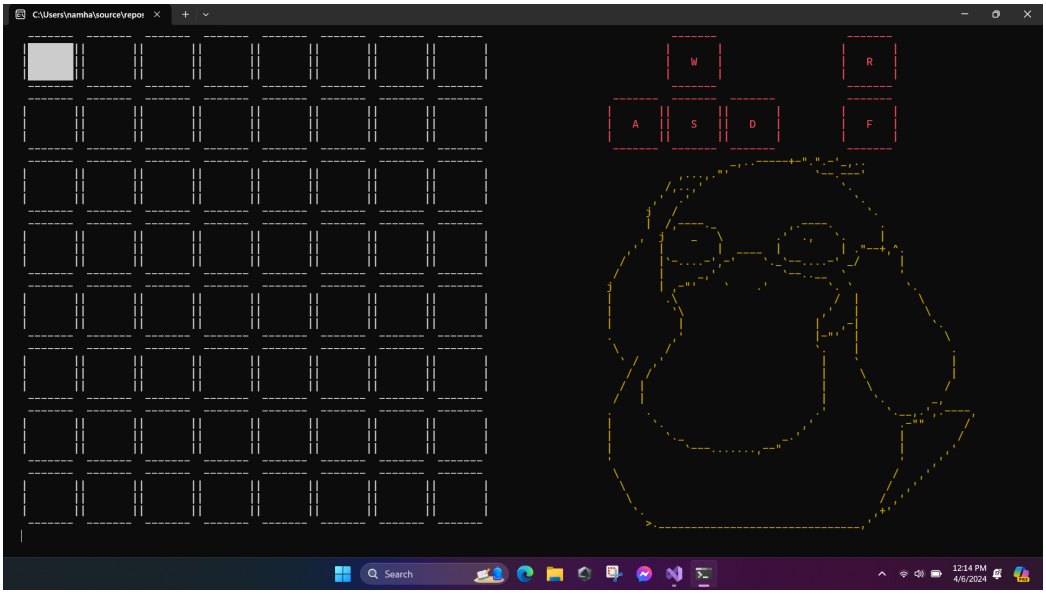


Figure 2.7: Interface of hidden mode [4]

2.2.7 Entering gift code

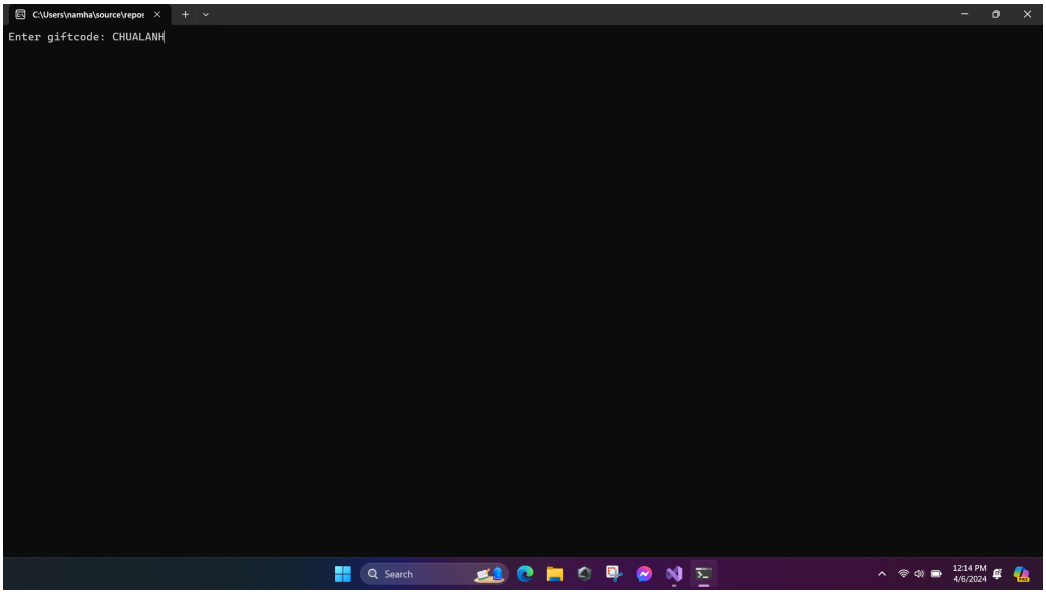


Figure 2.8: Gift code entry screen

- Each mode will have a different gift code.
- Each time a gift code is entered, the complexity of the game will be reduced by half.
- *Additionally, the game development team has added an interesting feature: *pressing a secret key on the keyboard* will remove 2 corresponding tiles.

2.2.8 Win

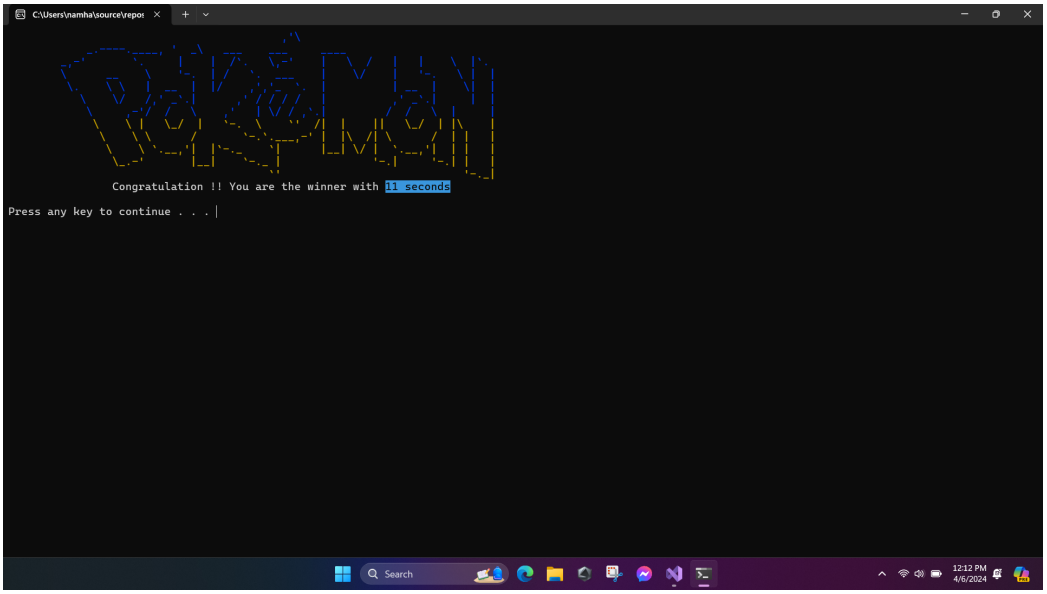


Figure 2.9: Victory screen [8]

- This is a genre of game with ranking rules based on the time to complete the game, NOT a genre of game based on the number of points achieved each time two corresponding tiles are found. Winning the game and the song "Legends Never Die" will be played. [9] ♫
- If you've completed the game and reached this screen, congratulations! You're a Pokémon and matching game enthusiast.

2.2.9 Ranking

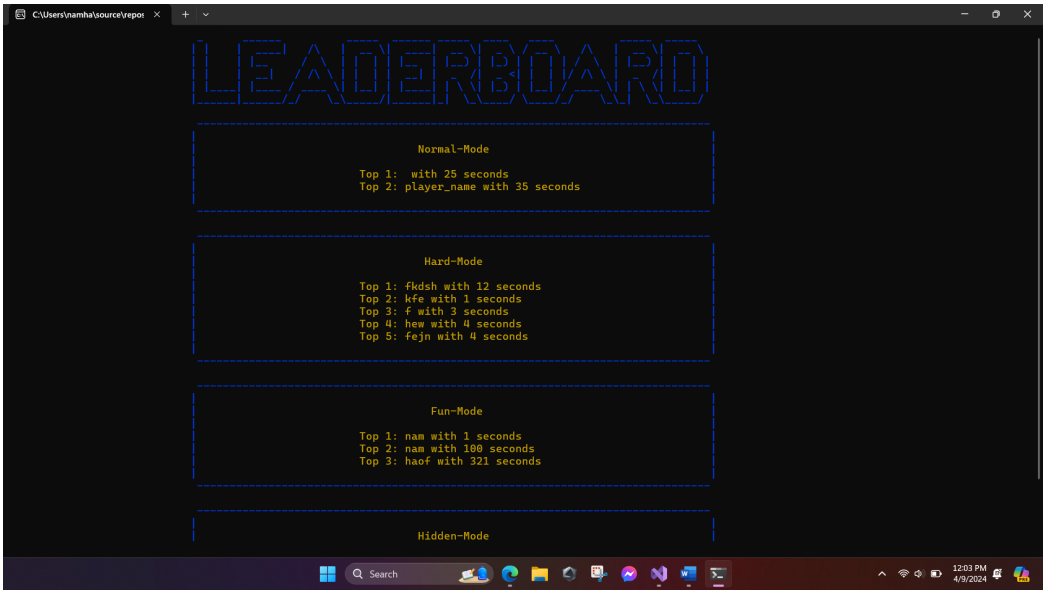


Figure 2.10: Ranking screen

- If there are no players, the ranking screen will display "No one plays this mode".
- Player-entered names will be displayed accurately even if they contain spaces.
- Show top 5 players have highest point.

2.2.10 End game

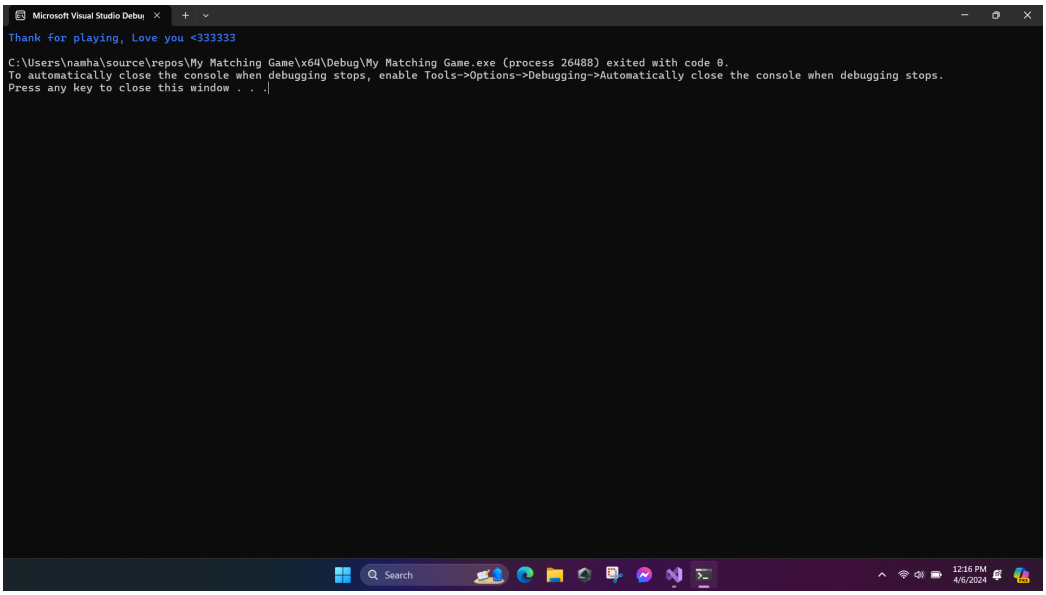


Figure 2.11: End game screen

- If you've reached this screen, thank you very much and see you again.

3 Code Explanation

3.1 Explain some structs' declaration and function

3.1.1 Structs' declaration

- `struct Storage`
  - `char name[100]`: array to store player name with maximum 99 characters (1 for '\0').
  - `int score`: to store the player score when the play ended.

- `struct Node`
  - `char** b`: 2D array to store the 2D array before deleting elements, use for Undo.
  - `Node* pNext`: Store the address of the next node.

3.1.2 Function

The general functions of the function have been outlined in the header files section.

3.2 Algorithm of Standard features

3.2.1 Starting the game

When the game starts, the logic begins with three functions in the Array.cpp file.

Algorithm 1 Function `void randomArr(char** a)`

```
1: procedure RANDOMARR(a)
2:     for i ← 1 to 8 do
3:         for j ← 1 to 8 do
4:             random ← random number from 0 to 63
5:             row ← random/8 + 1
6:             col ← random  mod 8 + 1
7:             swap a[i][j] with a[row][col]
8:         end for
9:     end for
10: end procedure
```

- **Purpose:** This function is used to generate a random 2D array by swapping elements randomly.

- **Details:** Iterate through each element in the 2D array. For each element, generate a random number in the range from 0 to 63. From this random number, calculate the corresponding row and column positions in the 2D array. Swap the value of the current element with the element at the calculated position. This will result in a random 2D array.

Algorithm 2 Function `void swapArr(char** a)`

```
1: procedure swapArr(a)
2:     for i ← 1 to 8 do
3:         count ← 0
4:         for k ← 1 to 8 − count do
5:             while a[i][k] is blank and k < 9 − count do
6:                 count ← count + 1
7:                 swap a[i][k] with a[i][9 − count]
8:             end while
9:         end for
10:    end for
11:    count ← 0
12:    for i ← 1 to 8 − count do
13:        check ← true
14:        while check and i < 9 − count do
15:            for k ← 1 to 8 do
16:                if a[i][k] ≠ blank then
17:                    check ← false
18:                    break
19:                end if
20:            end for
21:            if check then
22:                count ← count + 1
23:                if i < 9 − count then
24:                    for l ← 1 to 8 do
25:                        swap a[i][l] with a[9 − count][l]
26:                    end for
27:                end if
28:            end if
29:        end while
30:    end for
31: end procedure
```

- **Purpose:** This function is used to move blank characters to the end of each row and remove empty rows in the 2D array.

- **Details:** Iterate through each row of the 2D array. For each row, move blank characters to the end of the row. Then, remove empty rows by moving non-empty rows upwards, so that empty rows are shifted to the bottom of the 2D array.

Algorithm 3 Function `void gen2D(char** a)`

```
1: procedure GEN2D(a)
2:     for i ← 0 to 9 do
3:         for j ← 0 to 9 do
4:             a[i][j] ← blank
5:         end for
6:     end for
7:     temptemp ← 1
8:     for i ← 1 to 8 do
9:         for j ← 1 to 8 do
10:            if temptemp ≤ 10 then
11:                a[i][j] ← A
12:                temptemp ← temptemp + 1
13:            else if temptemp ≤ 20 then
14:                a[i][j] ← B
15:                temptemp ← temptemp + 1
16:            else if temptemp ≤ 30 then
17:                a[i][j] ← C
18:                temptemp ← temptemp + 1
19:            else if temptemp ≤ 40 then
20:                a[i][j] ← D
21:                temptemp ← temptemp + 1
22:            else if temptemp ≤ 48 then
23:                a[i][j] ← E
24:                temptemp ← temptemp + 1
25:            else if temptemp ≤ 56 then
26:                a[i][j] ← F
27:                temptemp ← temptemp + 1
28:            else if temptemp ≤ 64 then
29:                a[i][j] ← G
30:                temptemp ← temptemp + 1
31:            end if
32:        end for
33:    end for
34:    randomArr(a)
35: end procedure
```

- **Purpose:** This function is used to generate a 2D array of size 8x8, where characters from A to G are arranged continuously in order, and then the array is randomly shuffled.

- **Details:** Initialize the 2D array with all elements being blank characters. Set a variable temptemp to mark the order of characters from A to G. Iterate through each element in the 2D array and assign a value to each element based on the value of the temptemp variable, so that characters A to G are arranged continuously in order. Finally, call the randomArr function to randomly shuffle the elements in the 2D array.

3.2.2 Check Matching

Note: In the game, the checks will follow the order (I, L, Z, U), so when one check is successful, the other checks (if, else if, else) won't be executed. Therefore, if there is no successful check for I, then the check for L will be executed without needing to check for I, and similarly for Z and U.

The function checks for compatible tiles based on five functions in Check.cpp file.

```
bool checkI(char** a, int x1, int y1, int x2, int y2);
```

Algorithm 4 Pseudocode for `checkI` function

```
1: function CHECKI(a, x1, y1, x2, y2)
2:     if x1 = x2 then                                          ▷ If same row
3:         if y1 > y2 then          ▷ Ensure x1 and y1 are on the left to check only one case
4:             swap(y1, y2)
5:             swap(x1, x2)
6:         end if
7:         for i from y1 + 1 to y2 − 1 do                       ▷ Run from y1 to y2
8:             if a[x1][i] is not space then                    ▷ If there's a character, return false
9:                 return false
10:            end if
11:        end for
12:        return true
13:    end if
14:    if y1 = y2 then                                          ▷ If same column
15:        if x1 > x2 then          ▷ Ensure x1 and y1 are on the top to check only one case
16:            swap(x1, x2)
17:            swap(y1, y2)
18:        end if
19:        for i from x1 + 1 to x2 − 1 do                       ▷ Run from x1 to x2
20:            if a[i][y1] is not space then                    ▷ If there's a character, return false
21:                return false
22:            end if
23:        end for
24:        return true
25:    end if
26:    return false
27: end function
```

- There are only two cases when checking for I: they are either in the same row or in the same column. Therefore, there are two possibilities: `x1 == x2 or y1 == y2`. Otherwise, it will definitely be false.
- Case `(x1 == x2)`:
  - Because we want to check from left to right, `y1` must be less than `y2`. If it's greater, we'll swap `(x1, y1)` and `(x2, y2)`. Now, `(x1, y1)` is always on the left and `(x2, y2)` is always on the right, so we start checking.
  - Run a loop from `(x1, y1 + 1)` to `(x1, y2 - 1)` `(x1 == x2)`, checking if there's any character in between. If there is, return `false`; if not, return `true` after the loop.

  - If the two cells are adjacent, meaning `y1 + 1 == y2`, then obviously we won't run the loop and return `true` immediately.

- Case `(y1 == y2)`:
  - Because we want to check from top to bottom, `x1` must be less than `x2`. If it's greater, we'll swap `(x1, y1)` and `(x2, y2)`. Now, `(x1, y1)` is always on top and `(x2, y2)` is always at the bottom, so we start checking.
  - Run a loop from `(x1 + 1, y1)` to `(x2 - 1, y1) (y1 == y2)`, checking if there's any character in between. If there is, return `false`; if not, return `true` after the loop.
  - If the two cells are adjacent, meaning `x1 + 1 == x2`, then obviously we won't run the loop and return true immediately.
  - If neither of the above cases occurs, meaning there is no check for I, then return `false`.

```
bool checkL(char** a, int x1, int y1, int x2, int y2);
```

- For L check, there are also two main cases: either `(x1, y1)` is to the right and above `(x2, y2), or (x1, y1)` is to the left and above `(x2, y2)`. If it doesn't fit into these two cases, there is no L check.
- Case 1 `(x1 > x2 and y1 > y2) or (x1 < x2 and y1 < y2)`:
  - Because we only want to check one case where `(x1, y1)` is always on the upper-left, if `(x1 > x2 and y1 > y2)`, we'll swap `(x1, y1)` and `(x2, y2)`. Now, `(x1, y1)` is always less than `(x2, y2)`, so we start checking.
  - First, we check from left to right from `(x1, y1 + 1)` to `(x1, y2)` by checking I from `(x1, y1) to (x1, y2)` because I check only checks the middle segment, so we need to check if `a[x1][y2] == ' '`. If true, we'll check I from `(x1, y2)` to `(x2, y2)`, which means going downwards. If true, return `true`; if not, check other cases.
  - If the above case is not true, we'll check the case of going downwards first, then going from right to left. Check I from `(x1, y1)` to `(x2, y1)` and `a[x2][y1] == ' '`. If true, go from left to right by checking I from `(x2, y1)` to `(x2, y2)`. If true, return `true`; if not, exit and return `false`.
  - If both of the above cases are not true, return `false`, meaning there is no L check.
- Case 2 `(x1 > x2 and y1 < y2) or (x1 < x2 and y1 > y2)`:
  - Because we only want to check one case where `(x1, y1)` is always on the upper-right, if `(x1 > x2 and y1 < y2)`, we'll swap `(x1, y1)` and `(x2, y2)`. Now, `(x1, y1)` is always on the upper-right, so we start checking.
  - First, we check from top to bottom from `(x1 + 1, y1)` to `(x2 - 1, y1)` by checking I from `(x1, y1)` to `(x2, y1)` because I check only checks the middle segment, so we need to check if `a[x2][y1] == ' '`. If true, we'll check I from `(x2, y1)` to `(x2, y2)`, which means going from right to left. If true, return `true`; if not, check other cases.
  - If the above case is not true, we'll check the case of going from right to left first, then going from top to bottom. Check I from `(x1, y1)` to `(x1, y2)` and `a[x1][y2] == ' '`. If true, go from top to bottom by checking I from `(x1, y2)` to `(x2, y2)`. If true, return `true`; if not, exit and return `false`.
  - If both of the above cases are not true, return `false`, meaning there is no L check.
- If neither of the above large cases is true, return `false`.

`bool checkZ(char** a, int x1, int y1, int x2, int y2);`

- There are also two main cases for Z check: either `(x1, y1)` is to the right and above `(x2, y2)`, or `(x1, y1)` is to the left and above `(x2, y2)`. If it doesn't fit into these two cases, there is no Z check.
- Case 1 `(x1 > x2 and y1 > y2) or (x1 < x2 and y1 < y2)`:
  - Because we only want to check one case where `(x1, y1)` is always on the upper-left, if `(x1 > x2 and y1 > y2)`, we'll swap `(x1, y1)` and `(x2, y2)`. Now, **`(x1, y1)`** is always less than `(x2, y2)`, so we start checking.
  - Since there is already a function to check L, we'll utilize it to make Z check easier. First, we'll run a loop from `(x1, y1)` to the right incrementing `y1` but not equal to `y2`, and check if `(x1, y1 + some length)` is empty. If it's not, `break`; if it is, then we'll check L from that cell to `(x2, y2)`. If true, return `true`; if not, continue checking. If the loop finishes without returning true, then check the next case.
  - In the other case, we'll run a loop from `(x1, y1)` downwards incrementing `x1` but not equal to `x2`, and check if `(x1 + some length, y1)` is empty. If it's not, `break`; if it is, then we'll check L from that cell to `(x2, y2)`. If true, return `true`; if not, continue checking. If the loop finishes without returning true, exit.
  - If both of the above cases are not true, then there is no Z check.
- Case 2 `(x1 > x2 and y1 < y2) or (x1 < x2 and y1 > y2)`:
  - Because we only want to check one case where `(x1, y1)` is always on the upper-right, if `(x1 > x2 and y1 < y2)`, we'll swap `(x1, y1)` and `(x2, y2)`. Now, `(x1, y1)` is always on the upper-right, so we start checking.
  - First, we'll run a loop from `(x1, y1)` downwards incrementing `x1` but not equal to `x2`, and check if `(x1 + some length, y1)` is empty. If it's not, `break`; if it is, then we'll check L from that cell to `(x2, y2)`. If true, return `true`; if not, continue checking. If the loop finishes without returning true, then check the next case.
  - In the other case, we'll run a loop from `(x1, y1)` to the left decrementing `y1` but not equal to `y2`, and check if `(x1, y1 - some length)` is empty. If it's not, `break`; if it is, then we'll check L from that cell to `(x2, y2)`. If true, return `true`; if not, continue checking. If the loop finishes without returning true, exit.
  - If both of the above cases are not true, then there is no Z check.
- If neither of the above cases applies, return `false`, meaning there is no Z check.

`bool checkU(char** a, int x1, int y1, int x2, int y2);`

- Check U has 3 cases to examine: if they are in the same row, we only need to check the connecting line of U and U reversed; if they are in the same column, we check C and C reversed; if they are in different rows and columns, we check all 4: U, U reversed, C, C reversed.
- To facilitate U checking, I've created a 10 x 10 grid surrounded by spaces for easy path checking.
- Case `x1 == x2`:

  - We'll also utilize the L check function like in the Z check function.
  - First, we run a loop to go up from x1 and check if `(x1 - some length, y1)` is empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`; if not, continue checking.
  - Next, we'll run a loop downwards by incrementing `x1 (x1 + some length, y1)` and check if it's empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`.
  - If both cases fail ⇒ no U check.

- Case `y1 == y2`:
  - First, we run a loop to go left from y1 and check if `(x1, y1 - some length)` is empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`; if not, continue checking.
  - Next, we'll run a loop to go right by incrementing `y1 (x1, y1 + some length)` and check if it's empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return true.
  - If both cases fail ⇒ no U check.

- Case checking all 4 U, U reversed, C, C reversed:
  - Run all 4 loops as described above:
    * Run a loop to go up from `x1` and check if `(x1 - some length, y1)` is empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`; if not, continue checking.
    * Next, run a loop downwards by incrementing `x1 (x1 + some length, y1)` and check if it's empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`.
    * Run a loop to go left from y1 and check if `(x1, y1 - some length)` is empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`; if not, continue checking.
    * Next, run a loop to go right by incrementing `y1 (x1, y1 + some length)` and check if it's empty. If not, `break`; if empty, check L from there to `(x2, y2)`. If true, return `true`.
  - If all 4 cases fail, return `false`.

```
bool checkValidPairs(char** a);
```

- Run 4 nested for loops, where the first two loops hold the address of the initial cell `a[i][j]`, and the next two loops hold the address of the next cell `a[i][j + 1]`. If the cell is empty, skip it (continue) and check all 4 cases with 2 cells; if any of the cases are true, return `true`, otherwise continue increasing the address to the next cell. If the loop reaches the end, increment the first cell to the next and the next cell will be the next of the second cell. (brute force)
- If all 4 loops finish running and still no true result has been returned, it means there are no pairs connected, then return `false`.

3.2.3 Move Suggestion

Algorithm 5 Pseudocode for `moveSuggestion` function

```
1: function moveSuggestion(a, x, y)
2:     for i ← 1 to 8 do
3:         for j ← 1 to 8 do
4:             if a[i][j] is empty then
5:                 continue
6:             end if
7:             k ← i
8:             l ← j + 1
9:             if j = 8 then
10:                l ← 0
11:                k ← i + 1
12:            end if
13:            for k from k to 8 do
14:                l ← 1
15:                for l from l to 8 do
16:                    if checkI(a, i, j, k, l) and a[i][j] = a[k][l] and (i ≠ k or j ≠ l) then
17:                        repBoard(a, 10, 10, x, y, i, j, k, l)
18:                        return
19:                    end if
20:                    if checkZ(a, i, j, k, l) and a[i][j] = a[k][l] and (i ≠ k or j ≠ l) then
21:                        repBoard(a, 10, 10, x, y, i, j, k, l)
22:                        return
23:                    end if
24:                    if checkL(a, i, j, k, l) and a[i][j] = a[k][l] and (i ≠ k or j ≠ l) then
25:                        repBoard(a, 10, 10, x, y, i, j, k, l)
26:                        return
27:                    end if
28:                    if checkU(a, i, j, k, l) and a[i][j] = a[k][l] and (i ≠ k or j ≠ l) then
29:                        repBoard(a, 10, 10, x, y, i, j, k, l)
30:                        return
31:                    end if
32:                end for
33:            end for
34:        end for
35:    end for
36: end function
```

- Run 4 nested for loops, where the first two loops hold the address of the initial cell `a[i][j]`, and the next two loops hold the address of the next cell `a[i][j + 1]`. If the cell is empty, skip it (continue), otherwise, check all 4 cases with 2 cells; if any of the cases are true, assign these 2 cells as `(x1, y1)` and `(x2, y2)` (for coloring) and then return. If none of the cases are true, continue increasing the address to the next cell. If the loop reaches the end, increment the first cell to the next and the next cell will be the next of the second cell. (brute force)

3.2.4 Game finishing

The game will be considered complete when the function `bool checkEmpty(char** a);` (Check.cpp files) returns true value.

Algorithm 6 Function `bool checkEmpty(char** a)`

```
// Traverse the entire array to check if there is any non-empty character
for i from 1 to 8 do
    for j from 1 to 8 do
        if a[i][j] is not equal to ' ' then
            return false
        end if
    end for
end for
return true
```

1. **Initialization:** The function begins by traversing through each element of the 2D array ‘`a`‘, starting from row 1 and column 1, up to row 8 and column 8.
2. **Nested Loop Traversal:** Two nested loops are used to iterate over each element of the 2D array. The outer loop iterates over the rows (‘`i`‘), and the inner loop iterates over the columns (‘j‘). This ensures that each element of the array is visited.
3. **Checking for Empty Space:** At each position ‘`(i, j)`‘ in the array, the function checks if the value stored in ‘`a[i][j]`‘ is not equal to a space character `' '`. If a non-empty character is found at any position, the function immediately returns ‘`false`‘, indicating that the array is not empty.
4. **Returning Result:** If the function completes the traversal of the entire array without finding any non-empty characters, it returns ‘`true`‘, indicating that the array is empty.

3.3 Algorithm of Advanced features

3.4 Color Effect [1]

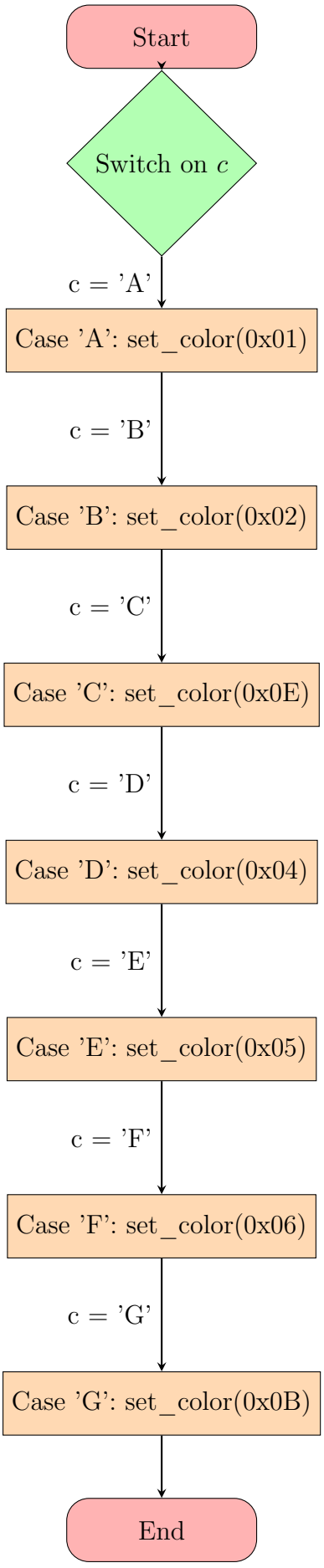
3.4.1 Function `set_color(code)`

Algorithm 7 Function `set_color(code)`

```
function SET_COLOR(code)
    color ← GetStdHandle(STD_OUTPUT_HANDLE)
    SetConsoleTextAttribute(color, code)
end function
```

The function `set_color(code)` sets the color for text displayed on the console screen.

- `color = GetStdHandle(STD_OUTPUT_HANDLE)` retrieves the handle of the console screen.
- `SetConsoleTextAttribute(color, code)` is called to set the color of the text displayed on the console screen. The parameter `code` represents the color code to be set.



The `repColor` flowchart illustrates the process of selecting corresponding colors based on a character value `c`. When a character `c` is passed into the `repColor` function, it checks the value of `c` and performs actions accordingly for each case.

- If `c` is 'A', the function sets the text color to the color specified by the color code 0x01.
- If `c` is 'B', the function calls the `set_color` function with the color code 0x02.
- Similar cases are handled for other `c` values, including 'C' through 'G', where each `c` value corresponds to a specific color code.

This allows the `repColor` function to adjust the text color based on the value of `c`, enabling the display of text in different colors depending on the value of the variable `c`.

Color list:

- 0x00: Black
- 0x01: Dark Blue
- 0x02: Dark Green
- 0x03: Dark Cyan
- 0x04: Dark Red
- 0x05: Dark Magenta
- 0x06: Dark Yellow
- 0x07: Gray
- 0x08: Dark Gray
- 0x09: Blue
- 0x0A: Green
- 0x0B: Light Blue
- 0x0C: Light Red
- 0x0D: Pink
- 0x0E: Yellow
- 0x0F: White

3.4.2 Sound effect

- The `PlaySound` [10] function is a built-in function in the `C++` programming language that allows developers to play sounds in their programs. This function is part of the Windows API (Application Programming Interface) and is therefore only available on Windows operating systems or when including the `<Windows.h>` library. While this function may not be able to play sounds simultaneously or handle MP3 files, it has the advantage of a quick response time. Therefore, we can use this function by converting audio files from MP3 to WAV format to play sounds when using the moving keys.
- List of sounds used: Sound files

Details: The PlaySound function in the game is placed in all game modes in the Game Mode.cpp file.

- Receive signals for selecting cells on the board:
  - When the player presses the number keys from 1 to 8 or the arrow keys to move the cursor on the board, the program will update the new position of the cursor based on the pressed key. There will be no sound played during the movement.

    ```
    PlaySound(NULL, NULL, SND_FILENAME | SND_ASYNC);
    ```

- Receive signals for pressing Enter to select a pair of cells:
  - When the player selects two cells on the board and presses Enter, the condition `if (getCommand == 13 && countTimes == 0 && a[x][y] != ' ')` will check if the player has selected a valid cell. If the condition is true, the program will save the position of the first cell to variables x1 and y1, and increase the countTimes variable by 1. When selecting the first cell, the sound "choose.wav" will be played.

    ```
    PlaySound(L"Music/choose.wav", NULL, SND_FILENAME | SND_ASYNC);
    ```

  - When the player selects another cell and presses Enter again, the condition `if (getCommand == 13 && countTimes == 1 && a[x][y] != ' ')` will check if the player has selected two valid cells. If the condition is true, the program will save the position of the second cell to variables x2 and y2, and continue to process this pair of cells.
- Check the selected pair of cells:
  - After receiving signals for selecting two cells and pressing Enter, the program will check if this pair of cells is valid. The functions `checkI()`, `checkL()`, `checkZ()`, and `checkU()` are called to check if the pair of cells forms a valid path.
  - If the pair of cells forms a valid path, the conditions in the if statements will be true, and the program will play the "mario.wav" sound to indicate that this pair of cells has been removed from the board. Otherwise, if no valid path is formed, the program will play the "buzzer.wav" sound to indicate that selecting these two cells is invalid.

    ```
    PlaySound(L"Music/mario.wav", NULL, SND_FILENAME | SND_ASYNC);
    PlaySound(L"Music/buzzer.wav", NULL, SND_FILENAME | SND_ASYNC);
    ```

- Check the end of the game:

- The game ends when there are no more pairs of cells on the board, checked through the `checkEmpty(a)` function.
- When the game ends, the program will play the "Legend.wav" sound to congratulate the player for winning.

```
PlaySound(L"Music/Legend.wav", NULL, SND_FILENAME | SND_ASYNC);
```

3.4.3 Visual effect

Function `void repBoard(char **a, int m, int n, int px, int py, int x1, int y1, int x2, int y2);` (from line 49 to 505 in the Draw board and pokemon.cpp file)

- Initialization: The function starts by setting up the console output color handle and defining a boolean variable check to keep track of whether certain parts of the board need special drawing treatment.
- Drawing the Game Board:
  - The function iterates through each row (i) of the game board, from 1 to 8.
  - For each row, it iterates through each column (j), from 1 to 8.
  - If the character at position (i, j) on the game board is not empty, it prints a series of dashes to represent the boundaries of the cell.
  - If the character is empty, it prints spaces to represent an empty cell.
  - Additionally, certain rows contain special decorative elements, such as buttons and decorative lines, which are drawn based on the row index (i).
- Drawing Special Elements:
  - The function draws the W, A, S, D, R, F buttons on the right side of the game board for rows 1 and 2. These buttons are drawn when the boolean variable check is true.
  - Decorative lines and text are drawn for rows 3 to 8 to enhance the visual appearance of the game board.
- Color and Highlighting:
  - Certain characters on the game board are highlighted based on their positions ((px, py), (x1, y1), (x2, y2)). If the current position matches any of these highlighted positions, the corresponding cell is highlighted with a different color.
- Console Output: The function uses cout statements to print the game board and its elements to the console screen.
- Termination: After drawing the entire game board, the function ends.

Function `void print_pokemon(int n_pokemon);` (from line 999 to 1539 in the Draw board and pokemon.cpp file)

Pokemon List: [4]

- In this function, the switch-case syntax is used to divide the selection cases on the game menu screen, and the SetConsoleTextAttribute function is used in each case to display the corresponding color interface for each case.
- Each case pointed to will display a Pokémon image as listed in the Pokémon list above.

3.4.4 Leaderboard

The source code for drawing the leaderboard is directly implemented in the Main.cpp file.

- Creating Frame and Title:
  - Before printing out the title and player list, the source code uses the '`SetConsoleTextAttribute()`' function to set the color for the characters that are about to be printed. This is done by passing a color code into this function, indicating the color to be set.
  - First, the source code creates a frame for the leaderboard by printing out a series of horizontal dashes. Then, it prints out the title "Normal-Mode" or "Hard-Mode" or "Fun-Mode" or "Hidden-Mode" depending on the mode the player has played.

- Printing Player List and Time:
  - After the color is set, the source code prints out the title of the leaderboard and the player list. Each line of the player list to be printed will have the color set previously.
  - The source code calls the '`readFile()`' function to read data from files containing information about players' scores in the corresponding game mode.
  - For each player and their time read from the file, the source code prints out a line containing the player's name and their time, while numbering them from 1 to n, depending on the number of players in the leaderboard.
- Finishing and Transitioning:
  - After printing the leaderboard, the source code uses the '`SetConsoleTextAttribute()`' function once again to set the color for subsequent messages, such as instructions or end-of-game notifications. This ensures that the messages thereafter will have the correct color.
  - Finally, the source code pauses the screen to allow the player to read the leaderboard information and then returns to the main menu of the game by calling the '`menuGame()`' function.

3.5 Speacial features

3.5.1 Undo

1. Check if pHead nullptr or not, if nullptr return
2. Traversal all over elements in array a and then assign array in pHead to array a.

Algorithm 8 undo($pHead$: Node pointer, a: pointer to pointer to char)

```
1: pCur ← pHead
2: if pHead is nullptr then
3:     return
4: end if
5: for i from 0 to 9 do
6:     for j from 0 to 9 do
7:         a[i][j] ← (pCur → b[i][j])          ▷ Copy the latest array to a
8:     end for
9: end for
```

Allow the player to move forward the steps that have been taken but have not Redo. The principle of operation is that before any two elements are removed, the 2D array is saved back to the Node and then addHead and LinkedList for storage. When Undo is requested, the game takes the stored array in the pHead assigned to the array used to print the game board and then removeHead to delete that array and move to the `phead->pNext`.

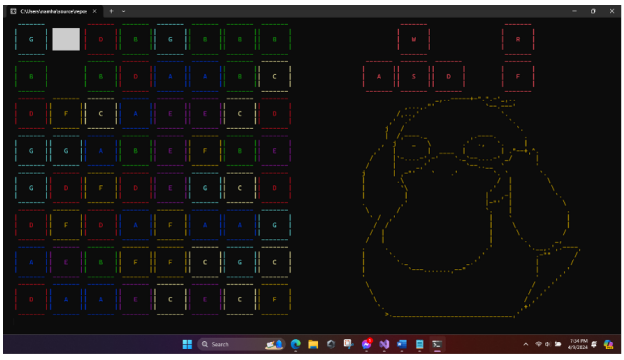


Figure 3.2: Before Undo

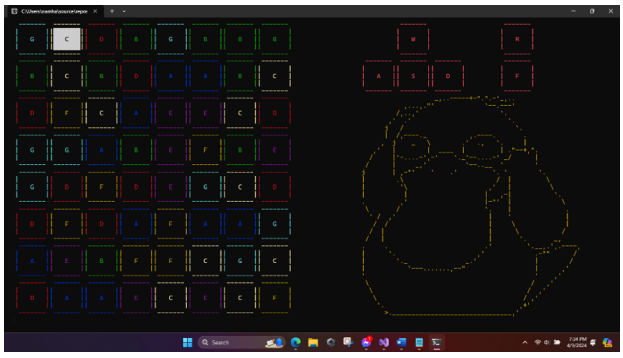


Figure 3.3: After Undo

3.5.2 Hidden mode

Function `void repBoardHidden(char** a, int m, int n, int px, int py, int x1, int y1, int x2, int y2);` (from line 506 to 998 in the Draw board and pokemon.cpp file)

1. Traversal all over the array
2. Draw box if that char is not ‘space‘
3. Only draw color and print char with x1, y1 and x2, y2 and draw white background with px, py.

In this mode, all the cells are white and the player will have to select each cell to know what it is and remember to match the same cells correctly. The game continues until all the cells are deleted and the game is over.

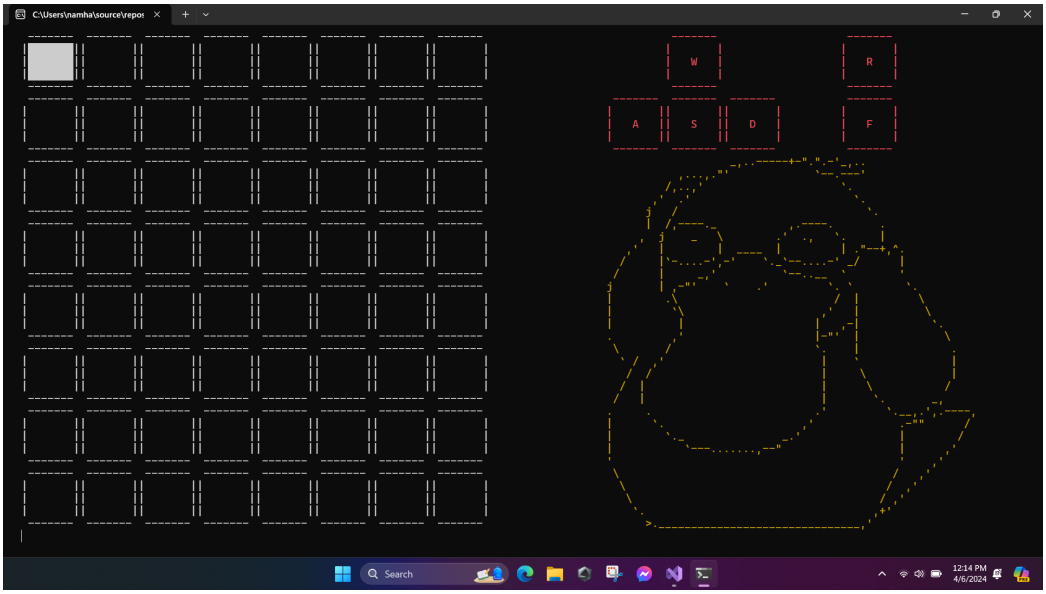


Figure 3.4: Interface of hidden mode

3.5.3 Delete and Shift up (use for build Hard game)

1. Traversal the 2D array (swap blank in 1 row)
2. If blank swap with last - count then count++ (first count = 0)
3. Traversal the 2D array (swap if 1 row is all blank)
4. Check if row is blank row
5. Swap with last row – count then count++

This function in the Array.cpp file will move all the blank spaces to the end of the array and replace the parts from the end to the top, so instead of merging the row, it will exchange values with the values of the end part to get a continious array. Especially when all characters of a row is blank, it will take the last row to replace the that one, exchange the values of the two rows together. If the last is a blank space, we're going to take the next to the last, and that's how it works until it's over.

Algorithm 9 Function `void swapArr(char** a)`

```
1: procedure swapArr(a)
2:     for i ← 1 to 8 do
3:         count ← 0                                   ▷ Initialize count of blanks
4:         for k ← 1 to 8 − count do   ▷ Iterate through elements, excluding blanks at the end
5:             while a[i][k] is ' ' and k < 8 − count do          ▷ Move blanks to the end
6:                 count ← count + 1
7:                 swap(a[i][k], a[i][8 − count])
8:             end while
9:         end for
10:    end for
11:    count ← 0                                       ▷ Initialize count of blank rows
12:    for i ← 1 to 8 − count do
13:        check ← true                  ▷ Initialize check for a row containing only blanks
14:        while check and i < 8 − count do
15:            for k ← 1 to 8 do
16:                if a[i][k] is not ' ' then          ▷ Check if row contains any character
17:                    check ← false                   ▷ Break loop if a character is found
18:                    break
19:                end if
20:            end for
21:            if check then  ▷ If the row contains only blanks, swap it with the last non-blank
    row
22:                count ← count + 1
23:                if i < 8 − count then
24:                    for l ← 1 to 8 do
25:                        swap(a[i][l], a[8 − count][l])
26:                    end for
27:                end if
28:            end if
29:        end while
30:    end for
31: end procedure
```

3.5.4 Collision Detection (use for build Hard game)

Function: `void undo(Node* pHead, char** a);` (Undo.cpp file)

This features is used for the hard mode. When you choose 2 wrong elements that they cannot match together, you the game will restore to the first array by calling Undo Funtion until pHead is nullptr so that you cannot undo this because the LinkedList will be removed all if you wrong.

Algorithm 10 Undo and Remove All Nodes

```
while pHead is not null do
    undo(pHead, a)
    removeHead(pHead)
end while
```

3.5.5 Gift code

1. Traverse the entire array.
2. Check all possible cases (I, L, Z, U) with the adjacent cell.
3. If true, delete the cells and return.

In our game, we offer different gift codes for each mode to enhance your gaming experience. To enter a gift code, press 'v' and then enter the respective code to delete half of the board by calling the `deleteSuggestion` function sixteen times.

- For Normal mode: `CHUALANH`
- For Hard mode: `KIENTHUC`
- For Fun mode: `KINHNGHIEM`
- For Hidden mode: `TRAINGHIEM`

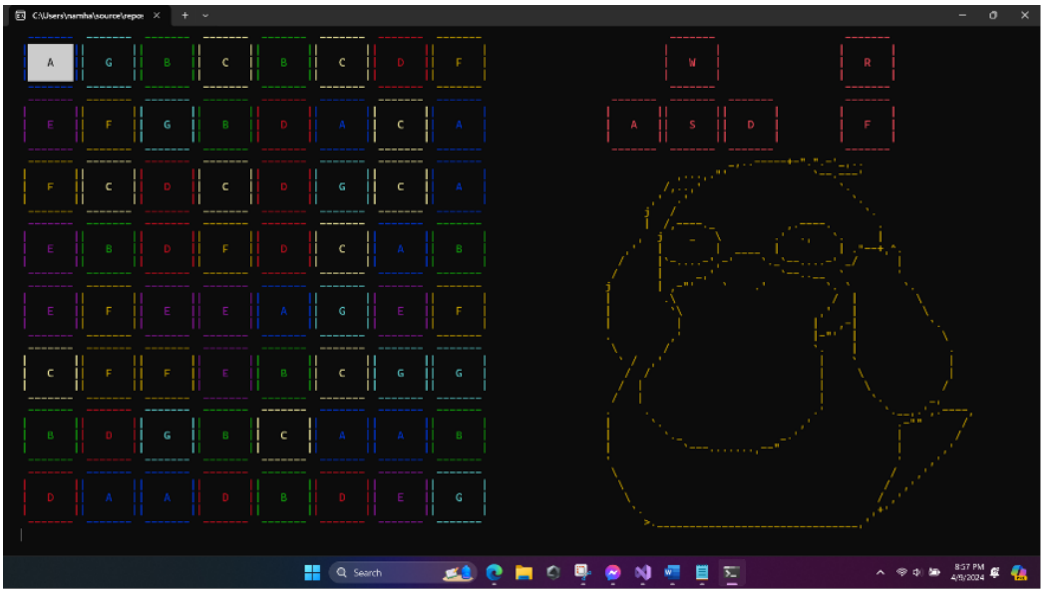


Figure 3.5: Before

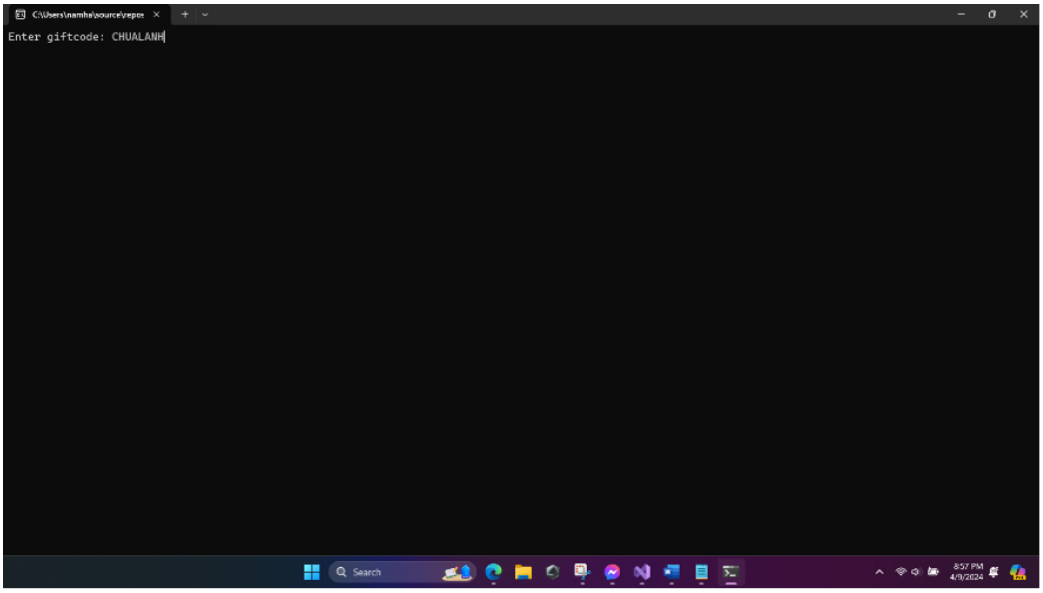


Figure 3.6: Enter giftcode

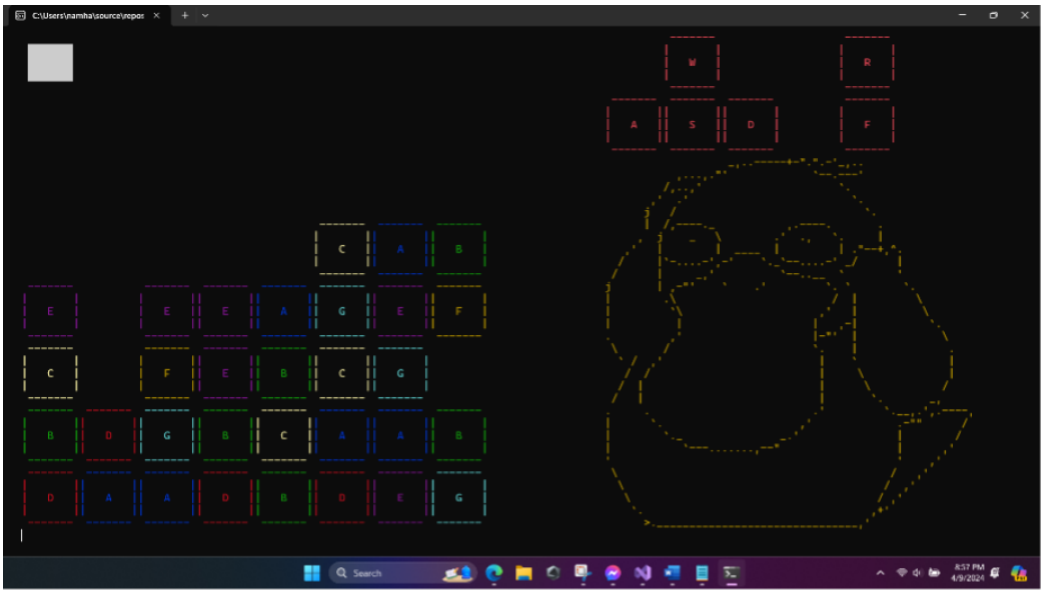


Figure 3.7: After

Algorithm 11 Delete Suggestion

```
1: procedure DELETESUGGESTION(a: array of pointers to characters)
2:     for i ← 1 to 8 do
3:         for j ← 1 to 8 do
4:             if a[i][j] equals ' ' then
5:                 continue
6:             end if
7:             k ← i
8:             l ← j + 1
9:             if j equals 8 then
10:                l ← 1
11:                k ← i + 1
12:            end if
13:            for k ← k to 8 do
14:                l ← 1
15:                for l ← l to 8 do
16:                    if checkI(a, i, j, k, l) and (a[i][j] equals a[k][l]) and not (i equals k and
    j equals l) then
17:                        a[i][j] ←' '
18:                        a[k][l] ←' '
19:                        return
20:                    else if checkZ(a, i, j, k, l) and (a[i][j] equals a[k][l]) and not (i equals k
    and j equals l) then
21:                        a[i][j] ←' '
22:                        a[k][l] ←' '
23:                        return
24:                    else if checkL(a, i, j, k, l) and (a[i][j] equals a[k][l]) and not (i equals k
    and j equals l) then
25:                        a[i][j] ←' '
26:                        a[k][l] ←' '
27:                        return
28:                    else if checkU(a, i, j, k, l) and (a[i][j] equals a[k][l]) and not (i equals k
    and j equals l) then
29:                        a[i][j] ←' '
30:                        a[k][l] ←' '
31:                        return
32:                    end if
33:                end for
34:            end for
35:        end for
36:    end for
37: end procedure
```

4 Project format explanation

4.1 Header files

These files are in "Source" folder.

- **Lib.h**:
  - This file serves the purpose of providing declarations and definitions for library functions used in the program.
  - It includes various standard C++ header files such as `<iostream>`, `<string>`, `<conio.h>`, `<Windows.h>`, `<fstream>`, `<cstring>`, and `<stdio.h>`.
  - These header files are essential for input/output operations, string manipulation, console input/output, Windows API functions, file input/output, and C-style string manipulation functions.
  - Additionally, it suppresses warnings for "_CRT_SECURE_NO_WARNINGS" specific to the MSVC compiler.
  - It includes the "Winmm.lib" library using the pragma directive.
  - The namespace declaration "using namespace std;" is used to avoid explicit qualification of standard library symbols.

- **Array.h**:
  - The purpose of the "Array.h" file is to declare functions related to manipulating 2D arrays, particularly for randomizing, generating, and swapping elements within the array. It includes declarations for the following functions:
    - **void randomArr(char** a)**: This function is responsible for randomly arranging the elements within the 2D array.
    - **void swapArr(char** a)**: This function swaps elements within the 2D array.
    - **void gen2D(char** a)**: This function generates a 2D array.

- **Check.h**:
  - The purpose of the "Check.h" file is to declare functions related to checking the validity of moves in the game. It includes declarations for the following functions:
    - **bool checkI(char** a, int x1, int y1, int x2, int y2)**: This function checks the validity of moves involving the "I" shape in the game.
    - **bool checkL(char** a, int x1, int y1, int x2, int y2)**: This function checks the validity of moves involving the "L" shape.
    - **bool checkZ(char** a, int x1, int y1, int x2, int y2)**: This function checks the validity of moves involving the "Z" shape.
    - **bool checkU(char** a, int x1, int y1, int x2, int y2)**: This function checks the validity of moves involving the "U" shape.

    - **bool checkValidPairs(char** a)**: This function checks if there are valid pairs of tiles that can be matched in the game.
    - **bool checkEmpty(char** a)**: This function checks if the game board is empty.

- **Draw board and pokemon.h**:
  - The purpose of the "Draw board and Pokemon.h" file is to provide functions related to drawing and formatting the game board and Pokemon characters. It includes declarations for the following functions:
    - **void set_color(int code)**: This function changes the color of the text and boxes of the cards to make identifying the cards easier.
    - **void repColor(char c)**: This function replaces the color of a character.
    - void **repBoard(char** a, int m, int n, int px, int py, int x1, int y1, int x2, int y2)**: This function replaces the game board with updated values.
    - **void repBoardHidden(char** a, int m, int n, int px, int py, int x1, int y1, int x2, int y2)**: This function replaces the game board with hidden values.
    - **void print_pokemon(int n_pokemon)**: This function prints Pokemon characters.

- **Game Mode.h**:
  - The purpose of the "Game Mode.h" file is to declare functions related to different game modes in the program. It includes declarations for the following functions:
    - **void EasyGame()**: This function represents the easy game mode.
    - **void FunGame()**: This function represents the fun game mode.
    - **void HardGame()**: This function represents the hard game mode.
    - **void HiddenGame()**: This function represents the hidden game mode.

- **Move Suggest and Delete.h**:
  - The purpose of the "Move Suggest and Delete.h" file is to declare functions related to suggesting moves and deleting suggestions in the game. It includes declarations for the following functions:
    - **void deleteSuggestion(char** a)**: This function deletes suggestions made for moves in the game.
    - **void moveSuggestion(char** a, int x, int y)**: This function suggests a move at a specified position on the game board.
    - **void moveSuggestionHidden(char** a, int x, int y)**: This function suggests a move at a specified position on the game board, considering hidden tiles.

- **ReadFile.h**:

  - The purpose of the "ReadFile.h" file is to declare functions related to reading and writing game results to and from files, specifically saving game results according to the top 5 fastest game completions for all 3 modes. It includes declarations for the following functions:
    * **void readFile(string name)**: This function reads game results from a file.
    * **void inputAndPrintFile(string namef, char a[100], int time)**: This function inputs and prints game results to a file.

- **Undo.h**:
  - The purpose of the "Undo.h" file is to declare functions related to managing a linked list used for undoing actions in the game. It includes declarations for the following functions:
    * **Node* createNode(char** temp)**: The pointer level 2 is stored in a new node that is created by this function for the linked list.
    * **void addHead(Node*& pHead, char** temp)**: This function adds a new node to the head of the linked list.
    * **void removeHead(Node*& pHead)**: This function removes the head node from the linked list.
    * **void removeAll(Node*& pHead)**: This function removes all nodes from the linked list.
    * **void undo(Node* pHead, char** a)**: This function assigns 2D array in pHead to a 2D array.

- **Struct.h**:
  - The purpose of the "Struct.h" file is to define structures used in the program. It includes definitions for the following structures:
    * **struct Storage**: This structure is used to store player information, including the player's name and score.
    * **struct Node**: This structure is used to define a linked list node for undoing actions in the game. It includes a 2D array to store data and a pointer to the next node in the linked list.

4.2 ".cpp" files

These files are in "Source" folder. They contain all definitions of the functions that are declared in the Header files and a "Main.cpp" contains the main() function.

4.3 Game result files

These files are located in the "Text" folder and consist of 4 files in .txt format aimed at storing game results and ranking for each play session:

- normal.txt: Storing game results of normal mode
- Hard.txt: Storing game results of hard mode

- fungame.txt: Storing game results of fun mode
- hidden.txt: Storing game results of hidden mode

4.4 Sound files

There are 5 ".wav" file located in "Music" folder. They are used to play sound when playing the game.

- sen.wav: The opening game sound [3]
- choose.wav: The sound when selecting a tile [6]
- buzzer.wav: The sound when failing to match a pair of tiles [5]
- mario.wav: The sound when successfully matching a pair of tiles [7]
- Legend.wav: The sound of the winning screen [9]

5 Program executing instruction

5.1 Compiler version

The program was built from this version: `g++ (MinGW.org GCC-6.3.0-1) 6.3.0`

5.2 How to compile

- All the source files are located in the "Source" folder.
- The program only uses libraries available in the standard `C++` library and some support libraries for the Windows operating system such as `<conio.h>` and `<Windows.h>`.
  - `Windows.h` is a header file in the Microsoft Windows operating system's API. It provides declarations and definitions for many functions, data types, and constants that are used in Windows programming. This header file allows developers to create Windows-based applications, access system services, handle graphical user interface (GUI) elements, manage processes, and more. Some of the functionalities provided by Windows.h include:
    - GUI programming using the Windows API (WinAPI).
    - Access to various system services such as file input/output, memory management, registry manipulation, etc. [11]
  - `conio.h` is a header file that provides functions for console input and output operations in C and C++ programming languages. It stands for "console input/output". This header file is primarily used in DOS and Windows environments to perform various console-related tasks, such as:
    - Reading input from the keyboard (`getch(), getche()`).
    - Writing output to the console (`putch(), cprintf()`).
    - Clearing the console screen (`clrscr()`).
    - Moving the cursor within the console window (`gotoxy()`).
    - Changing text color and background color.
    - And other console manipulation functions. [12]
- Compile and run the code using IDEs or integrated development environments such as Visual Studio Code, Visual Studio, DevC++, etc. Simply run it on the `main()` function or execute the following command:

  ```
  g++ Source/*.cpp -o <exe_file>
  ```

  Afterward, run the generated executable file:

  ```
  ./<exe_file>
  ```

- Furthermore, for the Windows operating system, the program will run when you "click" or execute the .exe file available in the project's workspace directory (My Matching Game.exe), but it's essential for the .exe file to remain within the project directory. If you simply move or separate the .exe file and run it independently, it may not be able to read or write files and play music.

6 Pointer and Linked List in project

6.1 Functions using Linked List

We have 4 functions related to linked list:

- `Node* createNode(char** temp);`
- `void addHead(Node*& pHead, char** temp);`
- `void removeHead(Node*& pHead);`
- `void removeAll(Node*& pHead);`
- `void undo(Node* pHead, char** a);`

This is a single linked list that has a node storing the 2D array and a pointer pNext. All these functions are used for storing the 2D array by adding head before deleting 2 elements so that we can undo the previous step by putting the array store in pHead to the main array, then remove pHead. In the hard mode, if the player chooses two wrong elements, the game will restore all the arrays in the linked list and then remove them all so that the player cannot undo that step and start again.

6.2 Pointer/LinkedList Comparison

- Pointer will consume less memory and help manage memory because it can be used in both heap and stack memory. Pointers can be accessed randomly through location indicators and easier to compare route tests. But it's hard for Pointer to delete the element.
- LinkedList can delete elements easily by changing the address of pNext. However, LinkedList is more memory-consuming by saving the address of the next node. Furthermore, LinkedList can only be accessed by traversal, making it difficult to check the path between the two cells.

⇒ Therefore, in this project we decided to use Pointer because it will consume less memory and will be able to check the path easily, but it will be difficult to delete the element, so we will fix it by assigning it as a 'space', to recognize when drawing the playboard. Otherwise, in Hard mode and Fun mode, using LinkedList can make it a lot easier to clear spaces between two cells than the algorithm we're using is to change positions with cells with letters at the bottom of a row or column.

7 Demonstration Video

Here is demonstration video on how to play.

References

[1] Nguyễn Thuận Phát. Matching-Game/Document at master · thuanphat611/Matching-Game (github.com), Source/board.cpp. *Matching-Game*, page 1, Oct 23, 2023.

[2] GEARVN. Đặc điểm nổi bật về game Pikachu kinh điển. *Hướng dẫn tải game Pikachu PC cổ điển*, page 1, April 7, 2024.

[3] cover Violin by Lindsey Stirling Kurousa-P. Senbonzakura. *Senbonzakura - cover Violin by Lindsey Stirling (youtube.com)*, November 7, 2020.

[4] Maija Haavisto MatheusFaria. 151 Pokemon ASCII Art (github.com), case 1, 2, 3, 4, 6, 54, 35. *151 Pokemon ASCII Art*, page 1, May 3, 2020.

[5] eritnhut1992. BUZZER OR WRONG ANSWER - Pixabay. *BUZZER OR WRONG ANSWER*, February 14, 2022.

[6] wolfy_sanic. ring. *collect ring - Pixabay*, January 31, 2022.

[7] collect ring. Mario sound in game. *Mario sounds*.

[8] Maija Haavisto. ASCII Art Pokemon - asciiart.eu, first case. *ASCII Art Pokemon*, page 1, April 7, 2024.

[9] Riot Games Music Team Alex Seaver of Mako and Justin Tranter. Legends Never Die. *ASCII Art Pokemon*, Oct 18, 2017.

[10] Max O'Didily. Youtube Link. *How to Stop and Play Music Using C++ (Simple) (PlaySound)*, March 3, 2023.

[11] 13 contributors of Microsoft. Windows API index. *Windows App Development*, March 15, 2023.

[12] 8 contributors of Microsoft. Console and port I/O. *C++*, October 26, 2022.