VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

FALCULITY OF INFORMATION TECHNOLOGY

Report

Project: Matching game

Course name: Programming Techniques

Student name:
Hieu, Ho Ngoc - 22127110
An, Nguyen Minh - 22127001

Advisor:
Bui Huy Thong
Tran Thi Thao Nhi

April 12, 2023

Contents

1 Introdution

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.

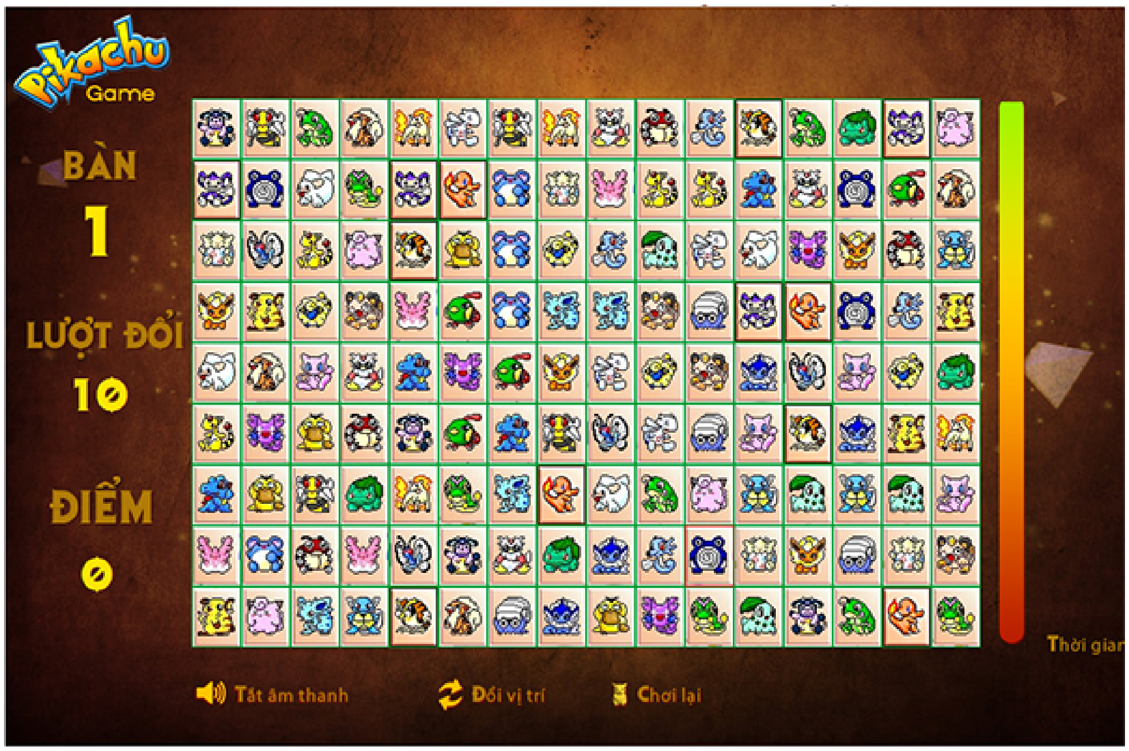


Figure 1: The Pikachu Puzzle Game (source)

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

2 How the game works

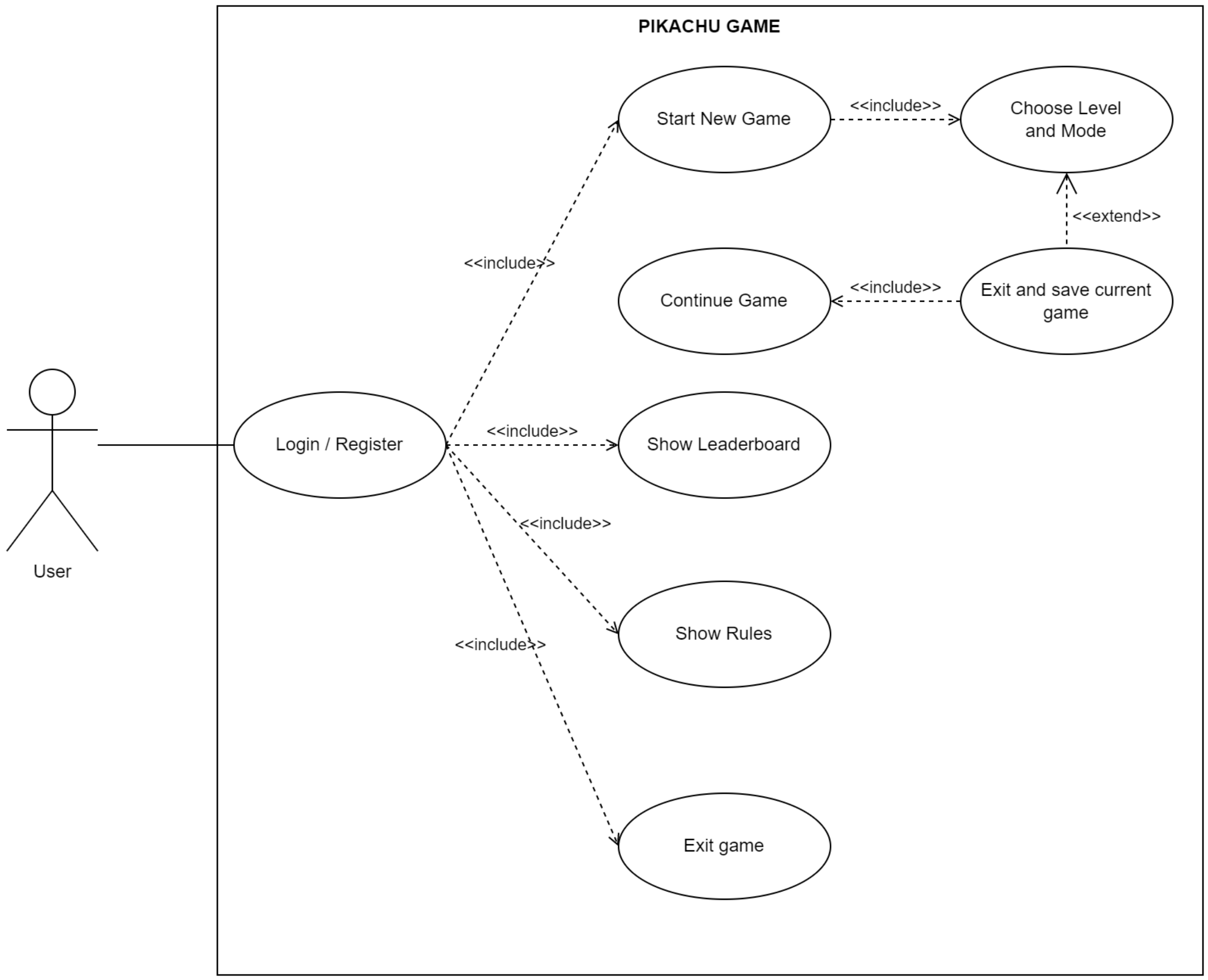
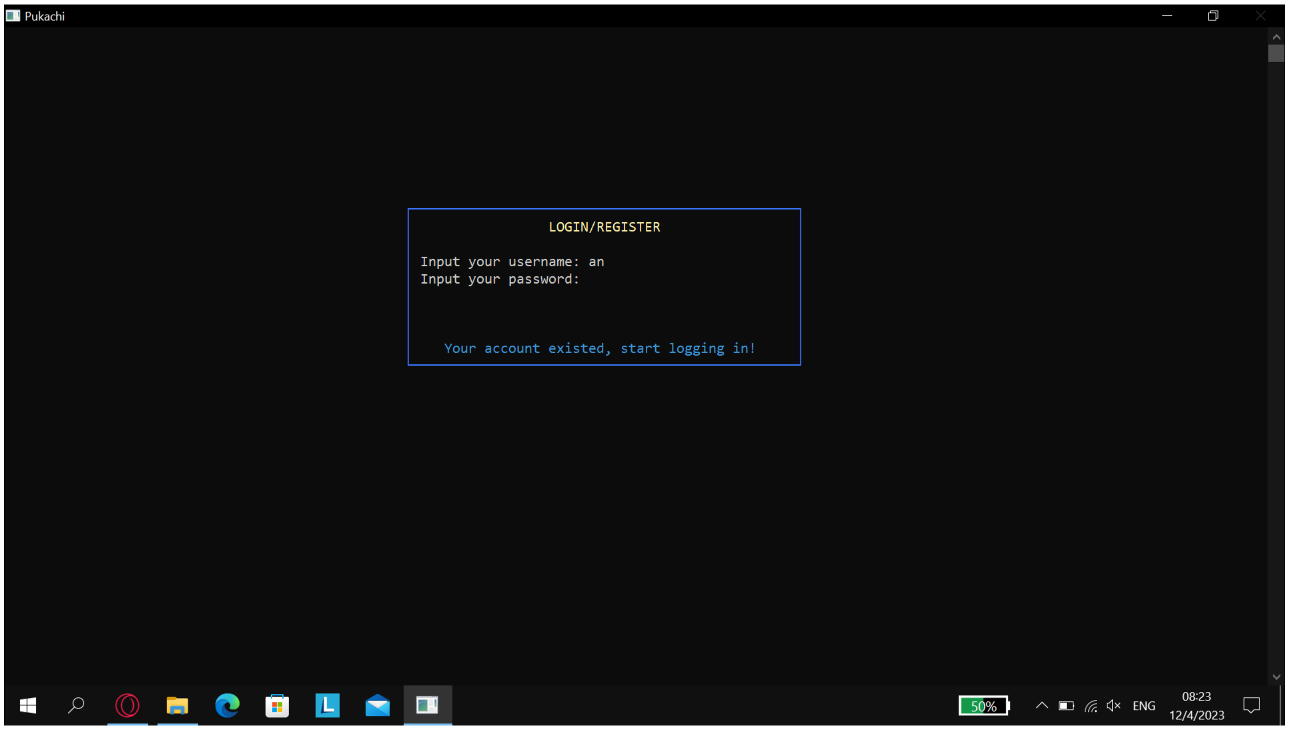


Figure 2: Use Case Diagram

2.1 Login/Register

- First, you input a username.



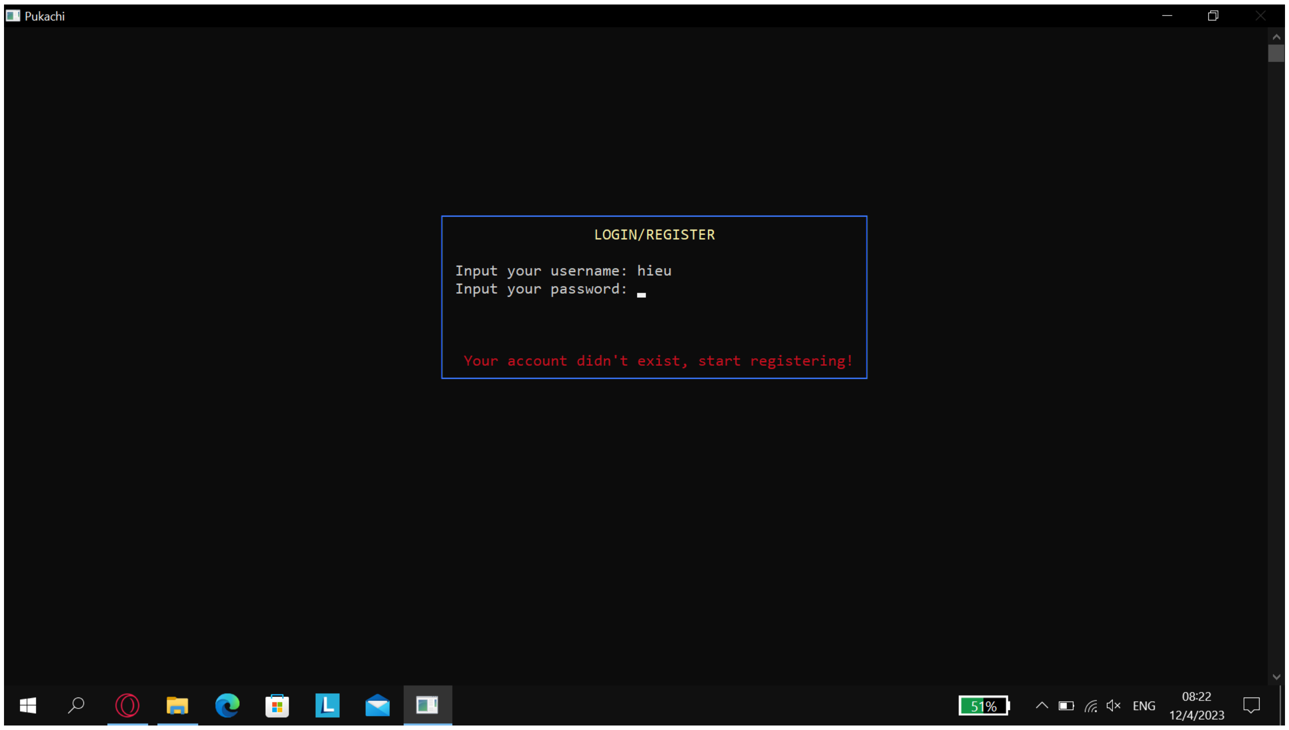
```
LOGIN/REGISTER

Input your username: an
Input your password:


Your account existed, start logging in!
```

Figure 3: Login

- If you have registered already, you will be asked to input your password. Else, you will have to create a password for your new account.



```
LOGIN/REGISTER

Input your username: hieu
Input your password:


Your account didn't exist, start registering!
```

Figure 4: Register

- After login/register successfully, you are switched to Menu Screen. Here is the navigation in the Menu Screen:

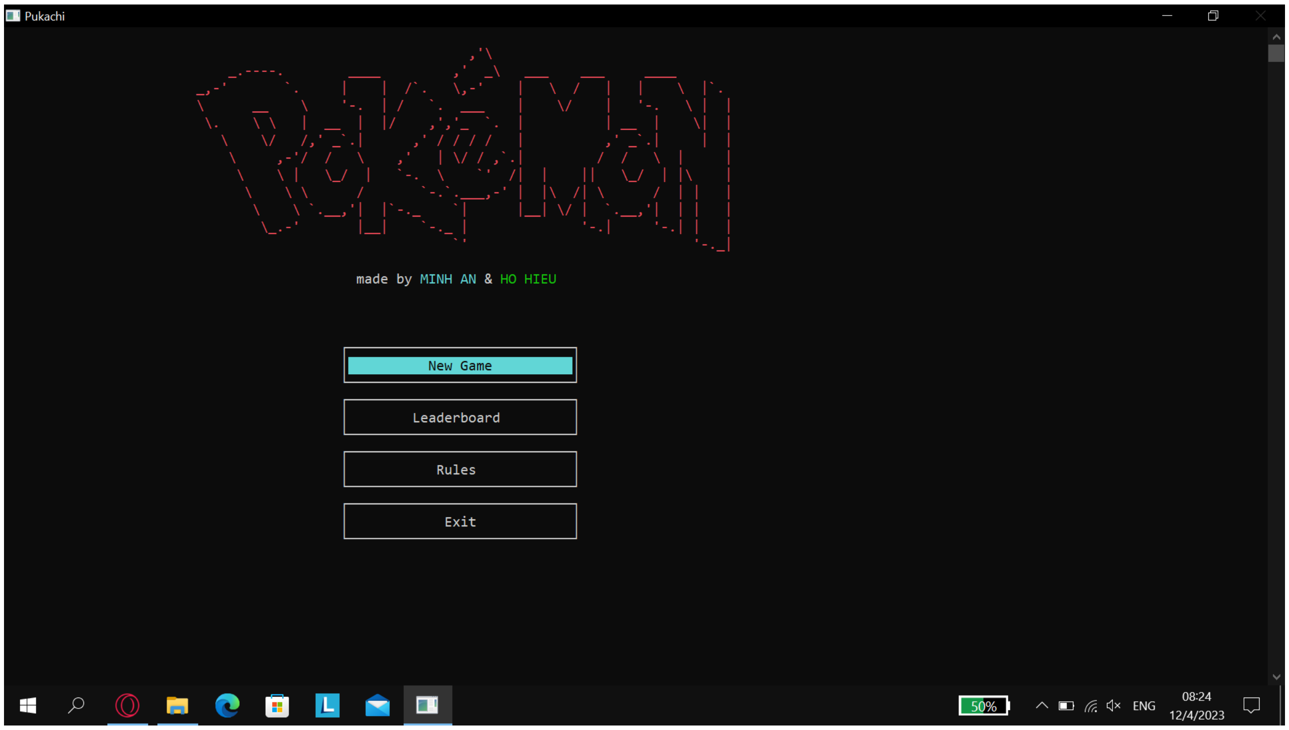


Figure 5: Main menu

  - Use W key or Up-arrow key to move cursor up.
  - Use S key or Down-arrow key to move cursor down.
  - Use Enter key to select.
  - Other keys are disable.

2.2 New game

After choosing New game, there is a Choose Level screen.Now, you have two options (Easy or Hard). Easy mode will create a 4x4 board and Hard mode is a 6x6 board.

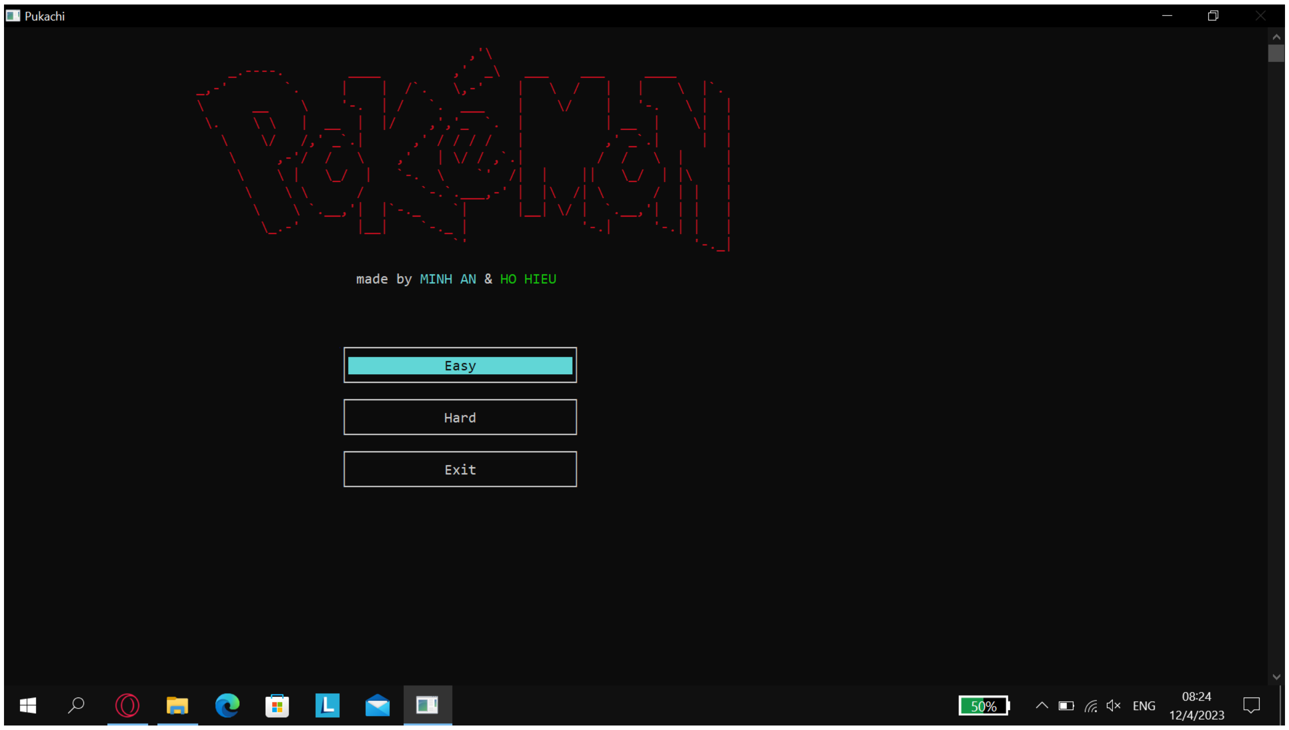


Figure 6: Choose level

2.2.1 Choose Mode

Then you choose the mode you want to play.

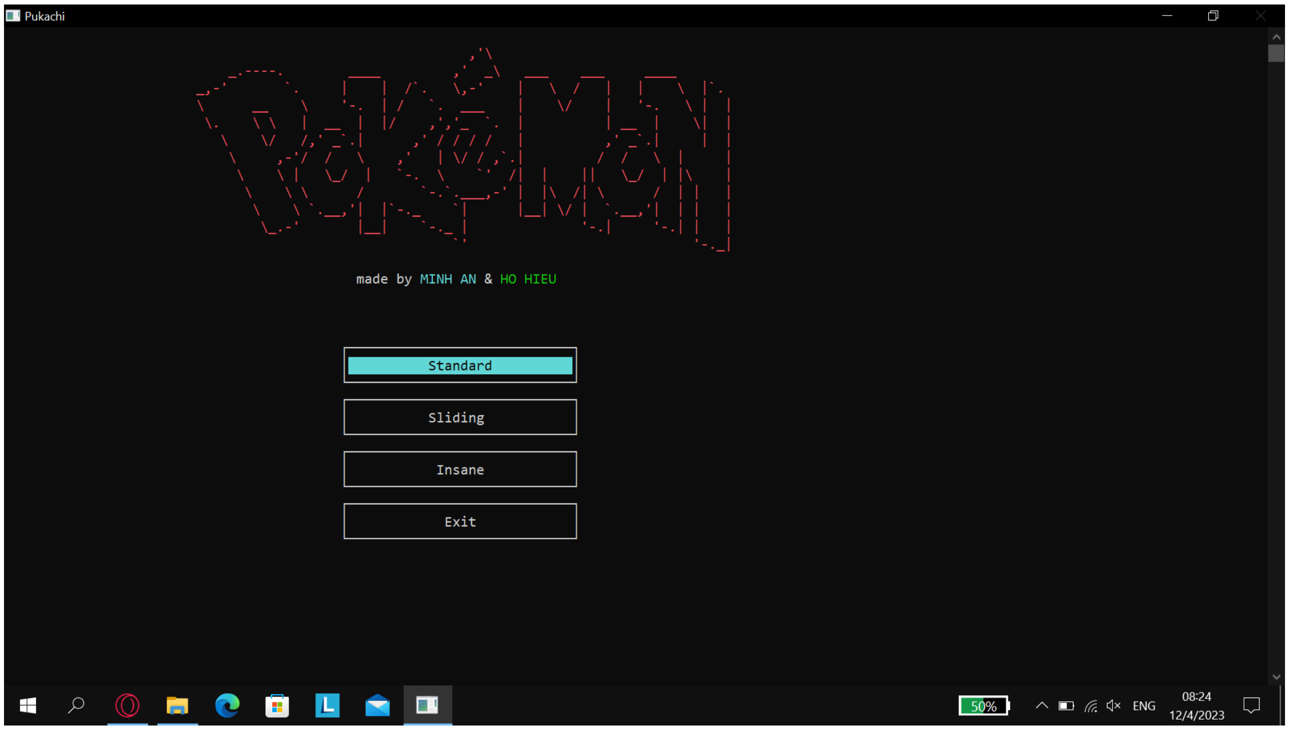
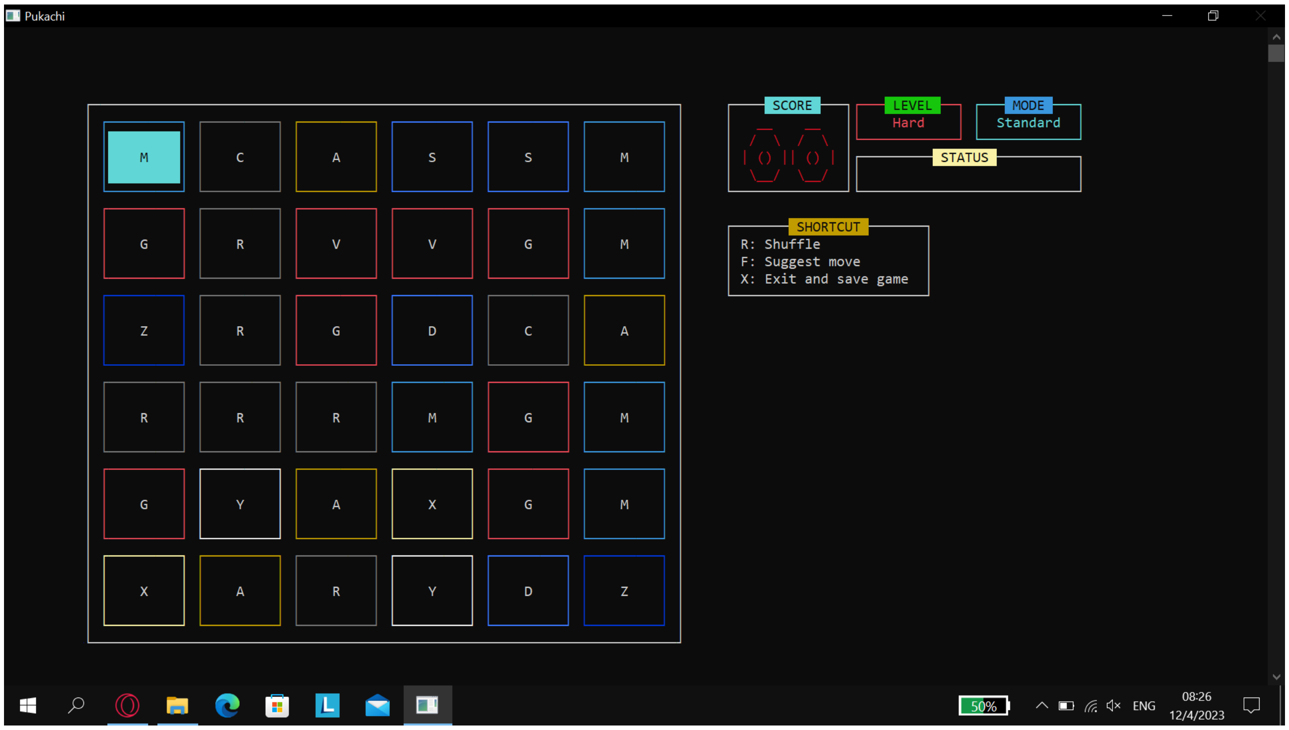


Figure 7: Choose mode

- Standard mode: This is the traditional mode of every Matching Game. There is a board with pairs of boxes, which have the same character and color. Your mission is to match these pairs of boxes following given patterns (see Rules and shortcuts).



LEVEL
Hard

MODE
Standard

SHORTCUT
```
R: Shuffle
F: Suggest move
X: Exit and save game
```

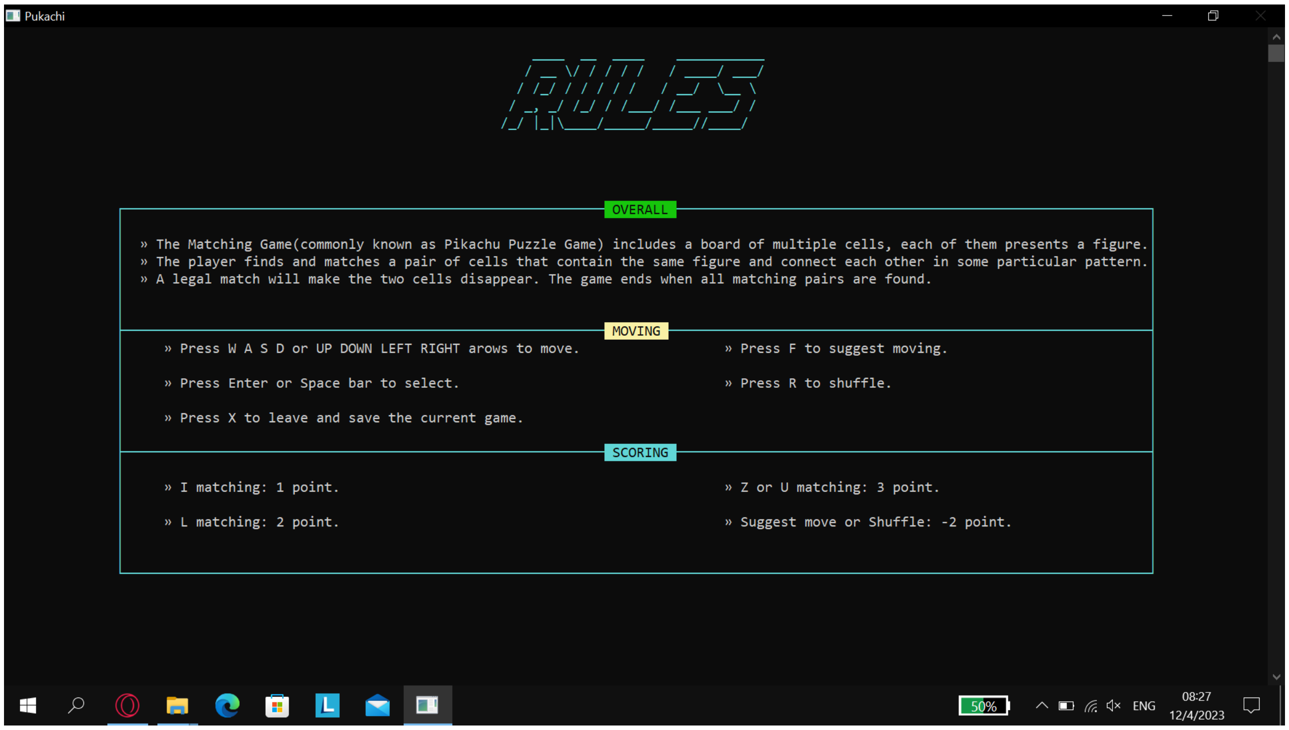
Figure 8: Standard mode

- Sliding mode
- Insane mode

2.2.2 Rules and shortcuts

- Use W A S D key or UP DOWN LEFT RIGHT arrow key to move cursor.
- Press Enter key to choose. Choosing one Character twice or choosing the disappear Character has no effect.
- If the pair you have chosen is not matching, the points will be decreased by 1.
- There are 4 types of matching:
  - I matching gives you 1 point.
  - L matching gives you 2 points.
  - U or Z matching gives you 3 points
- Press R to shuffle the board. The points will be decreased by 2.
- Press F to be suggested moving. The points will be decreased by 2.

- Press X to save game and exit.



```
OVERALL
» The Matching Game(commonly known as Pikachu Puzzle Game) includes a board of multiple cells, each of them 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.

MOVING
» Press W A S D or UP DOWN LEFT RIGHT arows to move.          » Press F to suggest moving.

» Press Enter or Space bar to select.                          » Press R to shuffle.

» Press X to leave and save the current game.

SCORING
» I matching: 1 point.                                         » Z or U matching: 3 point.

» L matching: 2 point.                                         » Suggest move or Shuffle: -2 point.
```

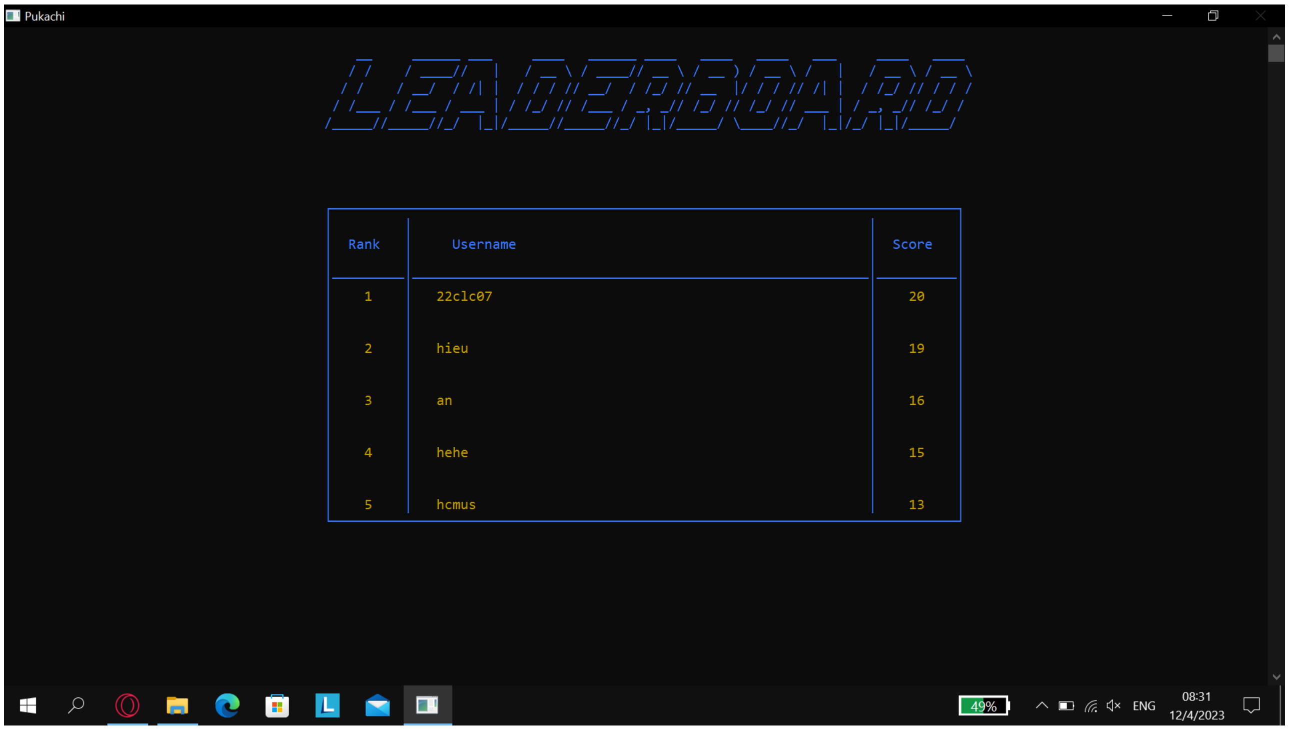
Figure 9: Rules screen

2.3 Continue game

Your latest game will be loaded (Includes: Remaining boxes, Score, etc.). If you have completed the previous game or have not played any game, this feature is disable.

2.4 Leaderboard

Show top 5 players have highest point.



```
Rank    Username                                Score

 1      22clc07                                  20

 2      hieu                                     19

 3      an                                       16

 4      hehe                                     15

 5      hcmus                                    13
```

Figure 10: Leaderboard screen

3 Code explaination

3.1 Explain some structs' declaration

3.1.1 Struct *Point*

The *Point* struct represents a coordinate point in a 2D-array. It has two fields:

- r: An integer representing the row number of the point.
- c: An integer representing the column number of the point.

3.1.2 Struct *Board*

Here's a breakdown of the different fields in the *Board* struct:

- $xBoardStart$ - an integer representing the x-coordinate of the top-left corner of the board.
- $yBoardStart$ - an integer representing the y-coordinate of the top-left corner of the board.
- $boxLength$ - an integer representing the length of each box on the board.
- $boxWidth$ - an integer representing the width of each box on the board.
- $size$ - an integer representing the size of the board.
- $pokeList$ - a pointer to a 2D integer array representing a square matrix. Each Pokemon is a number from 65 to 90, which is an ASCII value of an uppercase letter. An empty Pokemon is a space character, which is equal to 32.

3.1.3 Struct *GameInfo*

Here's a brief explanation of each field in the $GameInfo$ struct:

- *board*: A "Board" struct representing the game board.
- *score*: An integer representing the current score in the game.
- *selectedBlocks*: An integer representing the number of blocks that have been selected by the player.
- *remainBlocks*: An integer representing the number of blocks that still remain on the board.
- $p1$: A "Point" struct, representing the coordinates of the first selected block.
- $p2$: A "Point" struct representing the coordinates of the second selected block.
- *background*: A pointer to an array of strings representing the background image of the game.
- *mode*: An integer representing the current game mode.

3.1.4 Struct *Account*

The *Account* struct represents an account in a game. It has the following fields:

- *username*: A character array of length 20 representing the username of the account.
- *password*: A character array of length 20 representing the password of the account.
- *curScore*: An integer representing the current score of the player.
- *bestScore*: An integer representing the highest score achieved by the player on their account.
- *isPlaying*: A boolean value indicating whether the player is currently playing a game.
- *size*: An integer representing the size of the game board.
- *mode*: An integer representing the current game mode.
- *curPokeList*: A pointer to a 2D integer array representing the current state of the game board, where each cell can either be empty or contain a poke. The array has *size* rows and *size* columns and is updated after gameplay.

3.2 Standard Features

3.2.1 Starting the game

- **Function** *void randomPokemons(Board& board)*[1]

  This function is used to create a 2D array of characters randomly.

  1. To start, the function seeds the random number generator using the current time.

2. Initialize a variable called *size* to size of *board*.
3. Next, the function allocates memory for three arrays:
   - The first array, *pokemons*, is a 2D array with *size* rows and *size* columns that will temporarily store the characters representing the Pokemons for the game.
   - The second array, *checkDuplicate*, is a 1D array with $size \times size$ elements that will be used to keep track of whether a position on the board has already been assigned a Pokemon.
   - The third array, *pos*, is a 1D array with $size \times size$ elements that will store the randomly generated positions of the Pokemons on the board.
4. The function then generates random uppercase letters and assigns them to *pokemons* such that each Pokemon is assigned to two adjacent positions in the same row. This ensures that each Pokemon has a match on the board.
5. Next, the function generates random positions for each Pokemon by doing a loop and randomly assigning positions to the *pos* array until all elements of *pos* have been assigned a value. Inside the loop, there are three steps:
   - Enter a do-while loop that will generate a random number called *tmp* between 0 and $(size \times size - 1)$ and will keep looping until the element of *checkDuplicate* at index *tmp* is equal to 0.
   - After exiting the do-while loop, set the element of *checkDuplicate* at index *tmp* to 1.
   - Assign the value of *tmp* to the element of *pos* at the current loop index.
6. Finally, the function constructs the *pokeList* matrix of *board* by mapping the positions in *pos* to the corresponding elements in *pokemons* and assigning the corresponding Pokemon to that position on the board.

3.2.2 Check matching

- **Function** $Queue\ findPath(Board\ board,\ Point\ p1,\ Point\ p2)$[2]

  This function takes three arguments: a *Board* object representing a game board, and two *Point* objects representing the start and end points and then return the path between 2 points.

  1. Declare a constant integer variable *size* that represents the size of the board.
  2. Create a dynamic 2D array *e* with $(size+2)$ rows and columns and initialize each element to 0. This array will be used to represent the graph of the game board.
  3. Traverse the board and set the corresponding element in the *e* array to 1 if there is a Pokemon at that position, or 0 if there is not.
  4. Create an empty queue named *q* to store points that will be visited during the BFS algorithm.
  5. Create a dynamic 2D array *trace* with $(size + 2)$ rows and columns to store the trace of the path found by BFS. Initialize each element to {-1,-1}.

6. Add the end point to the tail of the queue q and set its trace value in the $trace$ array to {-2,-2}. Set the e values of the start and end points to 0.
7. While the queue is not empty, dequeue a point u from the front of the queue. If this point is the start point, break out of the loop.
8. For each of the four possible directions of movement, starting from point u, move in that direction as long as the next point is within the boundaries of the board and has a value of 0 in the e array. Add each new point visited to the tail of the queue q and set its trace value in the $trace$ array to u.
9. Once the BFS algorithm is completed, create an empty queue named res to store the path found.
10. If the trace value of the start point is not {-1,-1}, traverse the $trace$ array starting from the start point until the end point is reached, and add each point to the tail of the res queue.
11. Free the dynamically allocated memory for the e and $trace$ arrays.
12. Return the res queue containing the path found.

- **Funtion** $int\ checkMatching(GameInfo\&\ game,\ Queue\&\ path)$

  This function takes the $game$ object which contains the Pokemon board and the two selected blocks to check if they satisfy the conditions. After that, it assigns the path between them to a Queue named $path$ when they are matching.

  - This function creates a Queue named $path$ to contain the turning points of the path connecting 2 points (including the start and end points) by using the findPath function. We have to notice that the numbers of elements in a path for each matching condition are always between 2 and 4 points.
    * I Matching: 2 turning points.
    * L Matching: 3 turning points.
    * U and Z matching: 4 turning points.
  - Finally, return one of the below values:
    * Return -1: If 2 blocks are at the same row and column.
    * Return 1: If 2 blocks have the same Pokemons and the size of $path$ is between 2 and 4 elements (meet the conditions)
    * Return 0: If 2 blocks don't have the same Pokemons or the size of $path$ doesn't meet the conditions.

3.2.3 Game finishing

Check the following conditions:

1. Are there any cells left?

   - When starting the game, the variable remainBlocks in $GameInfo$ will be set to the total blocks of the game board.

- Every time the user chooses 2 blocks, the program will run the checkMatching function:
  - If *checkMatching* return 1: Start deleting those two pokemons by set their value to 32, which is the ASCII number of space character, and their positions will be replaced with the background by using drawBackground function. Finally, *remainBlocks* will be decreased by 2.
  - If *checkMatching* return -1 or 0: the variable *remainBlocks* will remain the same.

2. Are there any valid pairs left?

   - After every move, the game always checks if there are any valid pairs left by using moveSuggestion function. If not, the program will automatically shuffle the game board with shufflePokeList function and print the new board after that.

3.2.4 Save game

Reference: [3]

- **Function** $void\ inputAccountList(Account*\&\ account,\ int\&\ totalAccounts)$

  When opening the game, this function will start reading binary data from a file, decodes it using an XOR operation, and populates an array of Account structures with the decoded data. Here are the steps involved:

  1. Open a binary file named "saveGame.bin" in read-only mode.
  2. Assign the value of *totalAccounts* to the first integer in the file.
  3. Allocate memory for an array of Account structures with size equal to ($totalAccounts$+1). The last element in this array will be used if the user registers a new account.
  4. Loop through each account in the file and read its data into the corresponding element in the account array.
  5. Decode the data in the account structure for every single byte by performing an XOR operation with two mask characters.
  6. If the account is currently playing (i.e., its *isPlaying* field is set to 1), allocate memory for *curPokeList*, a 2D-array of integers to store the account's current Pokemon list.
  7. For each element in the *curPokeList*, read its data from the file and decode it using the same XOR operation as before.
  8. Close the file.

- **Function** $void\ outputAccountList(Account*\ \ account,\ int\ totalAccounts)$

  Every time the user returns to the main menu, this function will write data from an array of Account structures to a binary file, encoding the data using an XOR operation. Here are the steps involved:

  1. Open a binary file named "saveGame.bin" in write mode, overwriting any existing data.
  2. Write the *totalAccounts* variable to the file as an integer.
  3. Loop through each account in the account array.

4. Make a copy of the current account and encode its data for every single byte by performing an XOR operation with two mask characters.
5. Write the encoded account data to the file.
6. If the current account is currently playing (i.e., its *isPlaying* field is set to 1), loop through each element in its current Pokemon list and encode its data using the same XOR operation as before.
7. Write the encoded Pokemon list data to the file.
8. Close the file.

- **Function** void updateAccountAfterGame(Account& account, GameInfo& game, bool isPlaying)

  This function updates the account information of a player after a game, depending on whether the player is still playing or has finished playing. Here are the steps:

  1. Check if the player is still playing the game.
  2. If the player is still playing, update the account information with the current game's score, Pokemon list, game board size, and insanity level. Then delete the game's background to free up memory.
  3. If the player has finished playing, update the account information with the best score achieved so far and reset the current game score. Release the game's resources to free up memory and set the current Pokemon list to NULL.

3.3 Advanced Features

3.3.1 Color effect

- **Function** $void\ SetColor(int\ backgound\_color,\ inttext\_color)$ [4]
  - This function takes two arguments to set the background and the text color that will be printed to the console.
  - There are 16 colors, which are numbered from 0 to 15.
  - We combine this function with rand() **function** to make the game name ASCII art change color randomly.
  - We also use this function to generate the colors of character pairs to make them be recognizable.

3.3.2 Sound effect

- **Function** *mciSendString*[5] is a Microsoft Windows API function that can be used to play sound in C++. It allows a programmer to send a command string to the Media Control Interface (MCI) subsystem, which then carries out the command. Here is some advantages:
  - Can run simultaneously while playing other sound (suitable for playing background music).

  - Can play .mp3 files, which are typically 5 to 10x smaller than .wav files, so we use this function to play almost all sounds in our project.

  However, the drawback is its time response, so our solution is the following function.

- The *PlaySound* **function**[6] is a built-in function in the C++ programming language that enables 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. While the this function may not be able to play sounds simultaneously or handle MP3 files, it has the advantage of a quick response time. Because of this, we can use this function to play sounds when using the moving keys, which requires a rapid response.

- List of sounds used:
  - Background sound: Undertale - Megalovania
  - Selecting sound: Menu Game Button Click Sound Effect
  - Chose sound: Game Menu Select Sound Effect
  - Error sound: Error — Sound Effects (No Copyright)
  - Correct sound: Game Get Point - Sound Effects
  - Win sound: Undertale Sound Effect - You Win!

3.3.3 Visual effect

- ASCII art sources
  - Game name: asciiart
  - Other arts: Create ASCII text banners online

- Some functions to design UI
  - **Function** $CreateTextBox(int\ xStart,\ int\ yStart,\ int\ length,\ int\ width,\ string\ text)$
    This function draws a box at the given top-left coordinates with the given size and a string at the center.
  - **Function** $DrawBoardGame(GameInfo\&\ game,\ bool\ isSlow, bool\ isFlip)$
    This function takes $GameInfo$ object to draw a board game. $isSlow$ variable is used to decide whether draw slowly or not. $isFlip$ variable is used to check whether reveal the Pokemon or not in the Insane Mode.
  - **Function** *HighlightBox(int xStart, int yStart, int length, int width, string text, int mode)*
    This function has two mode. If *mode* = 1, it highlights the box at given specification. If *mode* = 0, it unhighlights that one.
  - Interactive functions [7]
    These functions use _getch function to catch user's navigation keys, change highlight boxes and change a *choose* variable. When user presses Enter key, the function returns *choose* to switch to the screen corresponding to the choose value.

- Draw matching lines
  If the checkMatching function returns 1, the game will draw the matching line. The idea is to traverse through each pair of nodes in "path" and draw a line between the coordinates of them.

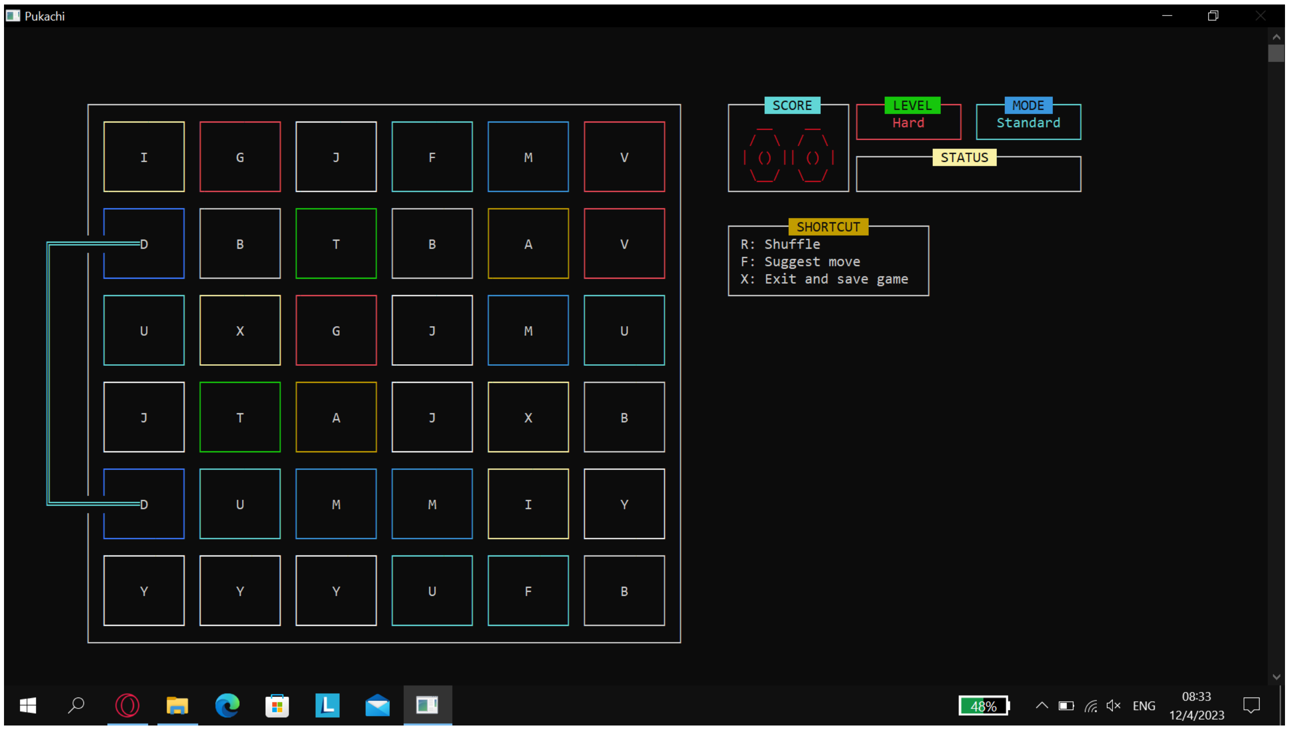


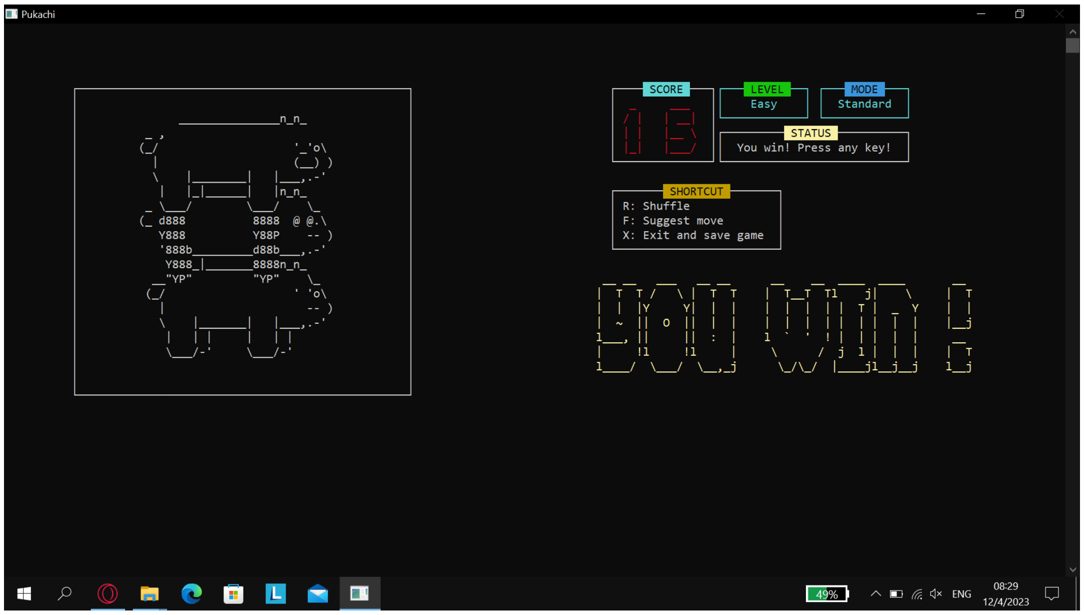
Figure 11: Draw matching line

  - **Function** $void\ drawMatchingLine(const\ GameInfo\&\ game,\ Queue\ path,\ bool\ isDraw)$
    This function takes in three parameters: a $GameInfo$ object, a queue named "path" to contain all the turning points by using findPath function, and a boolean value named $isDraw$. The function is responsible for drawing a line if $isDraw$ is equal to 1 and deleting that line if $isDraw$ is equal to 0.
    Here are the steps that the function performs:
    1. Initialize two node pointers, one for the current node and one for the previous node in the path queue.
    2. Loop through each node in the path queue and get the coordinates for the start and end points of the line segment to draw.
    3. Initialize some variables to keep track of the start of the line segment and the direction from which it came.
    4. Set the console color to a specific value for drawing the line.
    5. Loop through each point on the line segment and draw the appropriate character based on its location and direction.
    6. If the line segment is not the last one in the path queue, draw a special character at the end of the line segment to indicate its direction.
    7. Update the node pointers for the next iteration of the loop.
    8. Continue the loop until the function reaches the end of "path".

3.3.4 Background

When player starts the game, the program will generate a string array based on the game board size and assign it to the background array in *GameInfo* by using the function below:

- If size is 4, the program will load the "Easy.txt" file. (Ref: [8])

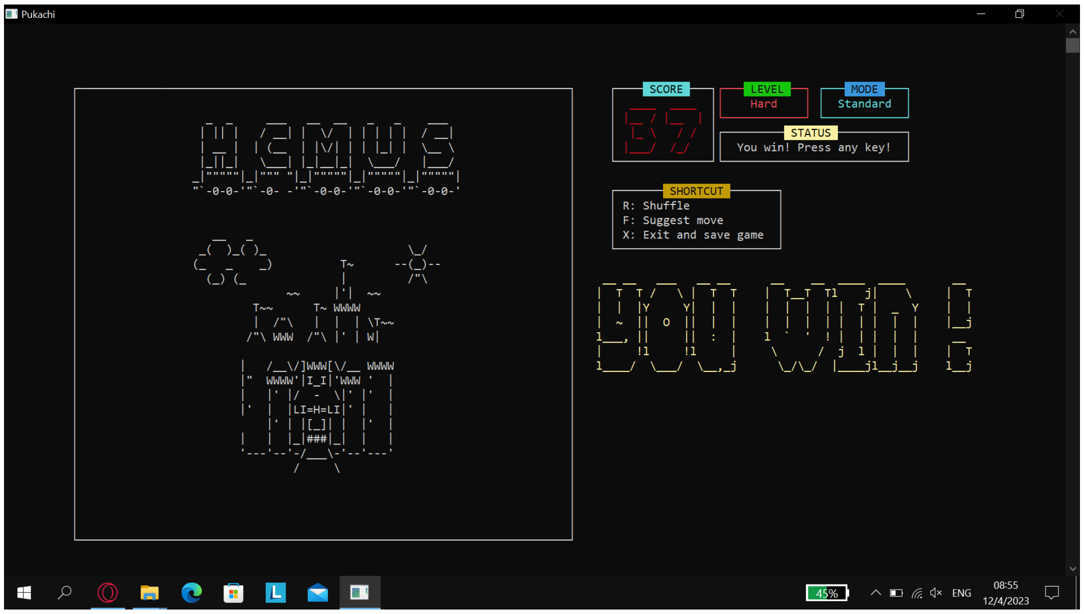


```
SCORE        LEVEL        MODE
             Easy         Standard
             STATUS
             You win! Press any key!

SHORTCUT
R: Shuffle
F: Suggest move
X: Exit and save game
```

Figure 12: Easy background

- If size is 6, the program will load the "Hard.txt" file. (Ref: [9])



```
SCORE        LEVEL        MODE
             Hard         Standard
             STATUS
             You win! Press any key!

SHORTCUT
R: Shuffle
F: Suggest move
X: Exit and save game
```

Figure 13: Hard background

- **Function** $string * \ createBackground(string\ fileName,\ const\ Board\&\ board)$

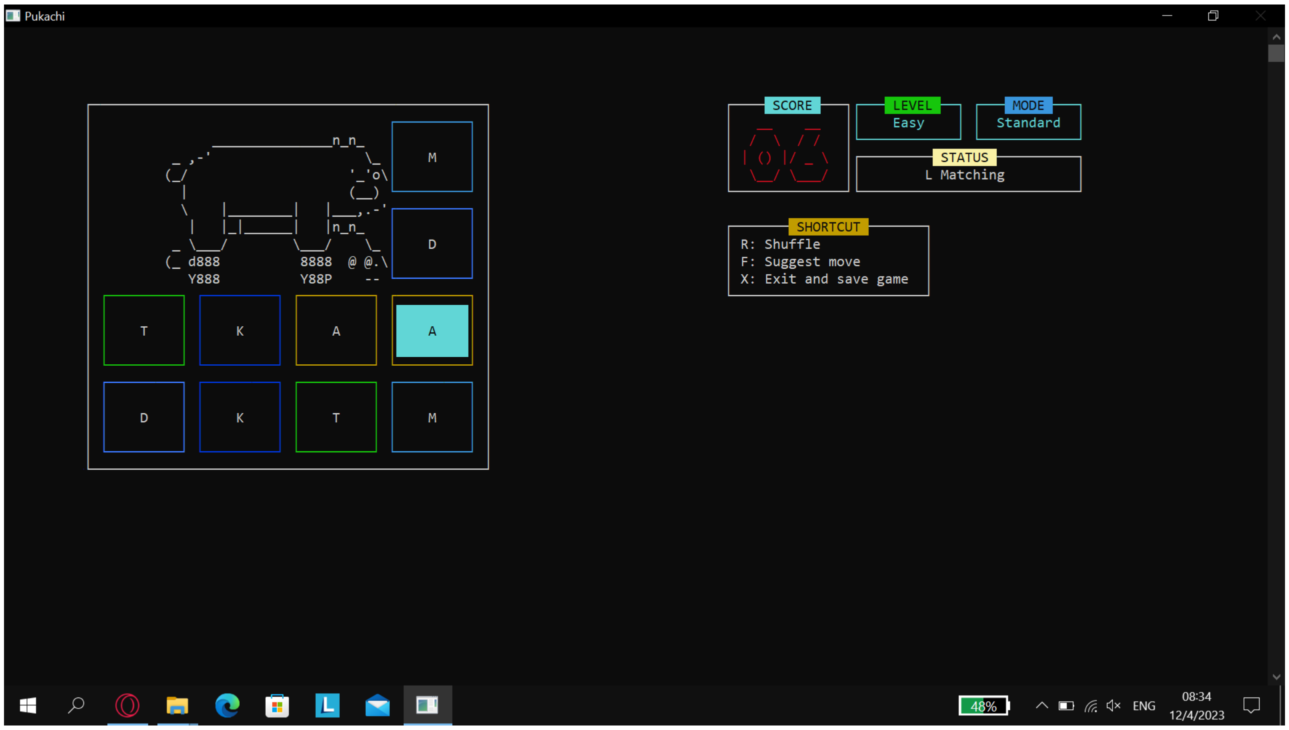
  This function reads in the contents of a text file and returns an array of strings, representing the background for a game board. Here are the steps:

  1. Open the file with the given file name.
  2. Create a new string array named $backgroud$. The size of the array is equal to the product of the board's size and box width, which means that the array will have enough elements to store the background for all boxes on the board.
  3. Loop through the file until the end of the file is reached.
  4. For each line in the file, store the line in an element of the $background$.
  5. Increment the counter variable to keep track of the current index in the array.
  6. Close the file.
  7. Return the $background$ array.

- **Function** $void\ drawBackground(GameInfo\&\ game,\ Point\ pokeIndex)$

  When two blocks are matching, the program will draw background content corresponding to those emptied cells. The function takes the $GameInfo$ object that contains the string array of background and the index of the Pokemon that we need to delete. Here are the steps:

  1. Calculate the dimensions of the box in which the background will be drawn based on the board dimensions.
  2. Calculate the starting point where the background should be drawn based on the dimensions of the box and the index of the Pokemon on the board.
  3. Determine the starting index of the background that should be drawn based on the index of the Pokemon.
  4. Loop through each row of the box and print each character of the background starting from the calculated start point and index of the background.



```
SCORE

LEVEL
Easy

MODE
Standard

STATUS
L Matching

SHORTCUT
R: Shuffle
F: Suggest move
X: Exit and save game
```

Figure 14: Background reveal

3.3.5 Leaderboard

If the user enters the "Leaderboard" option on the Main Menu, the program will run the function below and then show the Top 5 Players that finished their game with the highest score.

- **Function** $void\ sortDescendingAccountList(Account * \&\ account,\ int\ totalAccount)$

  This function takes an array of accounts and the total number of accounts as inputs. The function performs the Bubble sort algorithm to sort the accounts in descending order based on their best score.

3.3.6 Move suggestion

- **Function** $bool\ moveSuggestion(GameInfo\ game,\ Point\&p1,\ Point\&\ p2)$

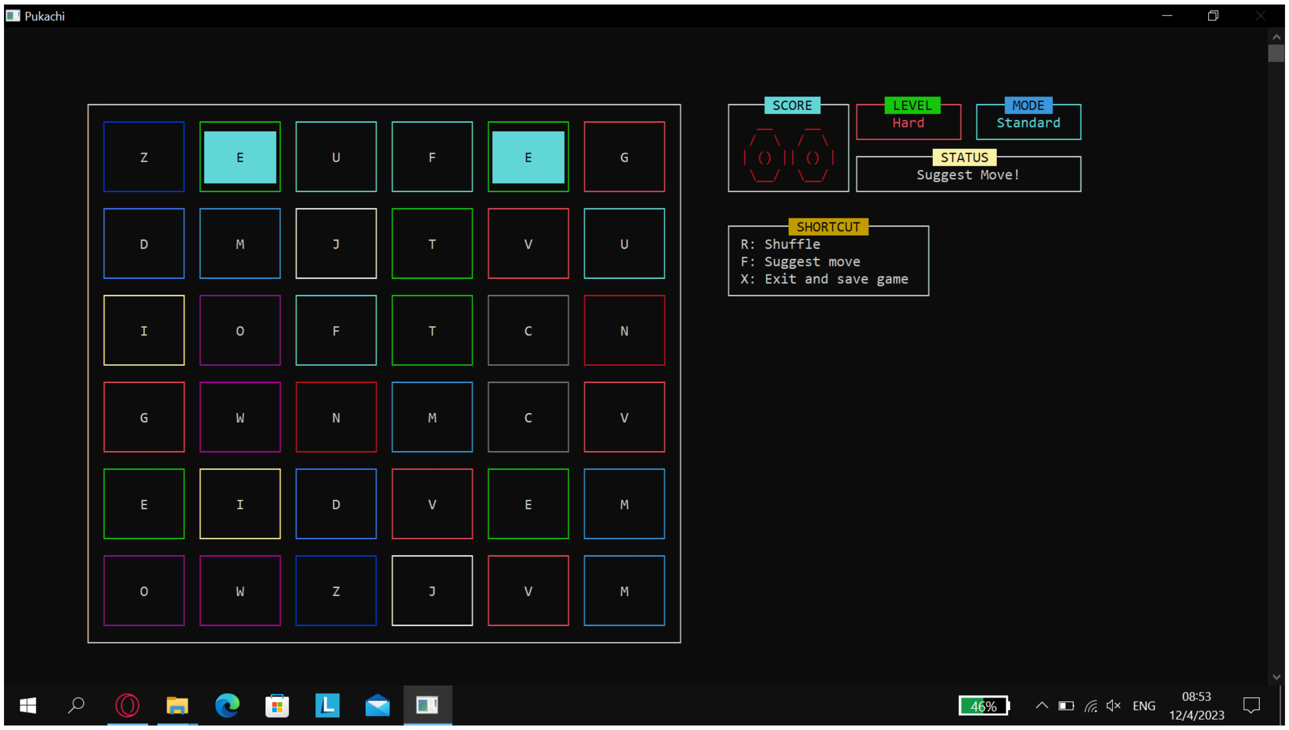
  The function takes in a $GameInfo$ object by value and two $Point$ objects, $p1$ and $p2$, by reference. If a matching pair of blocks is found, the function returns 1 then sets $p1$ and $p2$ to the index of those blocks. If not, it returns 0. Here is a step-by-step explanation:

  1. Set the $size$ variable to the size of the game board.
  2. Loop through all possible pairs of blocks on the game board.
  3. If either of the blocks is empty (represented by the ASCII value 32), continue to the next pair of blocks.
  4. Create an empty Queue object called $path$.
  5. Call the checkMatching function, passing in the $game$ object and the $path$ object by reference.

6. If *checkMatching* returns 1, set $p1$ to the coordinates of the first block in the matching pair and $p2$ to the coordinates of the second block, and return 1 to indicate that a valid pair has been found.
7. If no valid pair of blocks is found after looping through all possible pairs, return 0 to indicate failure.

- **Funtion** *void showSuggestMove(GameInfo& game)*

  The function takes a *GameInfo* object as input and tries to find a valid pair of blocks that can be matched by calling the moveSuggestion function. If a valid pair of blocks is found, the function highlights the boxes where the blocks are located on the board.



LEVEL
Hard

MODE
Standard

STATUS
Suggest Move!

SHORTCUT
R: Shuffle
F: Suggest move
X: Exit and save game

Figure 15: Show suggestion

3.4 Other features

3.4.1 Shuffle

Reference: [10]

- **Function** *void shufflePokeList(GameInfo& game)*

  This function randomly shuffles the positions of all characters that are not deleted.

  Here are the steps of the function:

  1. Seed the random number generator with the current time.
  2. Create an array *nonDeletedIndices* to store the indices of non-deleted Pokemon on the board and initialize a counter *numNonDeleted* to 0.

3. Loop through each cell of the board and check if the cell contains a Pokemon. If so, it stores the index of that cell in the *nonDeletedIndices* array and increments the counter *numNonDeleted*.
4. It uses the Fisher-Yates shuffle algorithm to shuffle the *nonDeletedIndices* array. The algorithm works by starting at the end of the array and swapping the current element with a randomly selected element from the remaining unshuffled elements.
5. Loop through each cell of the board again and check if the cell contains a Pokemon. If so, it retrieves the next shuffled index from the *nonDeletedIndices* array and calculates the new row and column indices corresponding to the shuffled index.
6. Swap the Pokemon at the current cell with the Pokemon at the shuffled cell.
7. Deallocate the *nonDeletedIndices* array.

3.4.2 Sliding mode

- How to play: In Standard Mode, when a matching pair is found, the game simply deletes those blocks. Unlike that, in Sliding Mode, it moves all the blocks in the same row from the right side of the deleted pair to the left. The scoring is the same as Standard mode (see Rules and shortcuts).

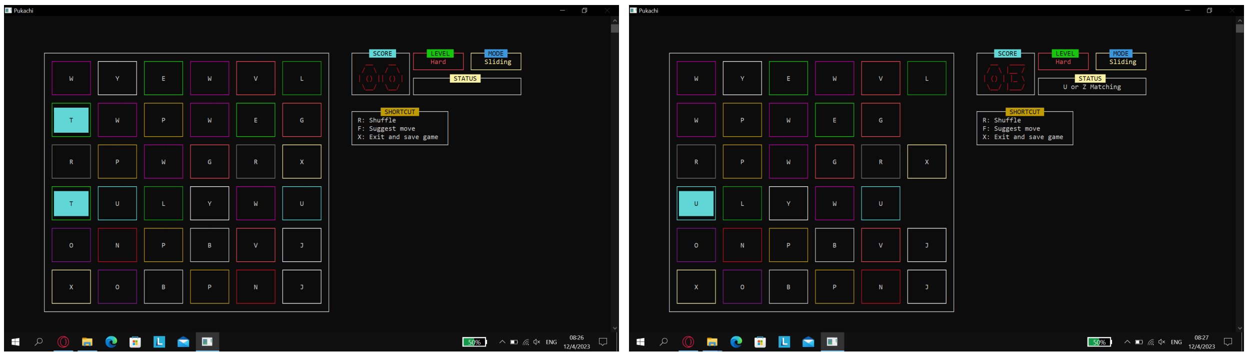


Figure 16: Sliding mode

- How to implement this mode: To do this, it copies their values to the column to their left in the same row, starting from the position of the deleted pair and ending at the second-to-last column. Finally, it sets the block at the last column to 32. After this, it draws the updated board using a function called drawBoardGame.

3.4.3 Insane mode

- How to play: This is an additional feature, based on the idea of Memorize Game. The characters and colors are hidden most of the time you play. You only peek the whole board at the beginning of the game or after you choose a pair of boxes. The scoring is the same as Standard mode (see Rules and shortcuts).

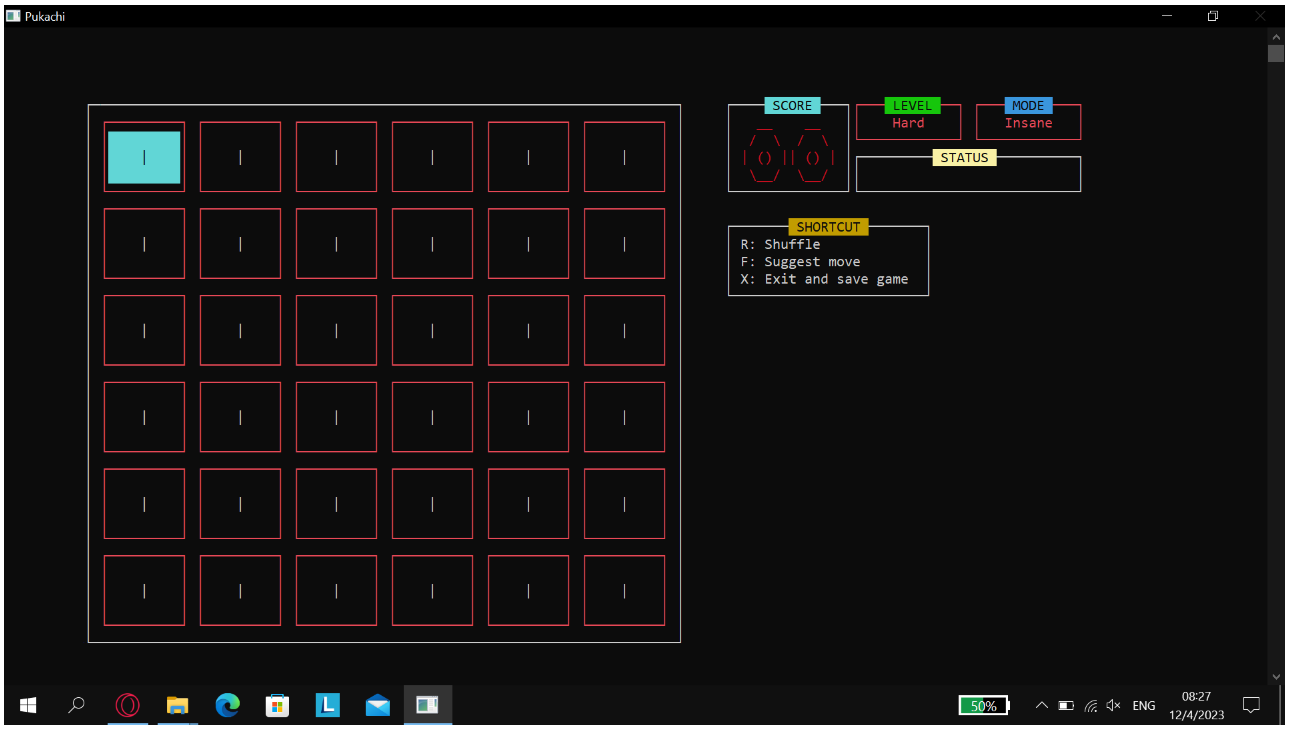


Figure 17: Insane mode

- How to implement this mode: The key idea of this mode is to change the value of the *isFlip* variable in the drawBoardGame function to reveal the characters when necessary. The game shows the hidden board only at the beginning of the game or after a pair of boxes has been chosen by calling the drawBoardGame function with *isFlip* set to true. After that, the game will pause for 1 second to give the player time to remember the characters, and then hide the entire board by calling the drawBoardGame function with *isFlip* set to false.

4 Program executing instruction

4.1 G++ version

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

4.2 How to compile

- All the source files locate in the "Source" folder.
- The program uses the mciSendString() function provided by the Windows Multimedia Library. As a result, we need to add "-lwinmm" when compiling with g++ to link the Windows Multimedia Library with the program.
- For example, user can build a ".exe" file named "Pukachi.exe" by using this:

  g++ Source/*.cpp -lwinmm -o Pukachi

- To help these steps become easier, user can run a batch file located in the folder named "BuildAndRun.bat" to automatically build and run a file named "Pukachi.exe".
- The folder also contains an executable file that was built before named "Pukachi.exe". User can run this file to play the game immediately.

5 Project format exlaination

5.1 Header files

These files are in "Source" folder.

- Account.h: This file contains the declaration of the below things:
  - Struct Account.
  - The functions that are used to change the Account list and the Binary saving file.
- Board.h: This file contains the declaration of the below things:
  - ASCII art of numerical digits by using a character array.
  - Struct Board.
  - randomPokemons function.
  - The functions that are used to display the interface when playing.
- Console.h: This file contains the declaration of the below things:
  - The functions that are used to change the setting of console.
  - The functions that are used to play sound.
- Game.h: This file contains the declaration of the below things:
  - Struct GameInfo and Point.
  - Struct *Node* and *Queue* that are used for the check-matching algorithm.
  - The functions that are used to operate the game when playing.
- Menu.h: This file contains:
  - Definition of a set of constants that represent different colors.
  - Definition of a set of constants that represent different keyboard keys.
  - Declaration of the functions that are used to create visual effects for boxes.
  - Declaration of the functions that are used to display the Menu screens (Main menu, Choose Level menu, etc.).

5.2 ".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.

5.3 Background files

There are 2 text files named “Hard.txt” and “Easy.txt” located in “Background” folder. They are used to display the background of the game.

5.4 Save Game file

There is a binary file named “saveGame.bin” located in “SaveGame” folder. It is used to save the information of the account list.

5.5 Sound files

There are 5 “.mp3” files and 1 “.wav” file located in “Sound” folder. They are used to play sound when playing the game.

6 Demonstration video

Here is demonstration video on how to play.

References

[1] Louis Tran. Function BoardView::buildBoardData(). *Pikachu-Game*, April 22, 2022.

[2] PhongLMH. Function Board::findPath(int _x, int _y, int x, int y). *Code Game Challenge in C++ in 24 hours (part 2)*, March 5, 2020.

[3] Nguyen Hoang Khang. Student at HCMUS. ID - 22127178.

[4] J.Delta. Link. *Windows.h and the console content format function*, June 25, 2020.

[5] Microsoft. link. *mciSendString function*, June 6, 2016.

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

[7] Thien Tam Nguyen. Link. *Techniques for creating dynamic menus in C++*, April 27, 2021.

[8] ASCII Veni, Vidi. Redraw this art. *Tweet post*, April 8, 2021.

[9] Louis Tran. medium.txt. *Pikachu-Game*, April 22, 2022.

[10] Wikipedia. Link. *Fisher–Yates shuffle*.

A Appendix

Some console setting functions [4]

- void setAndCenterWindow();
- void DisableResizeWindow();
- void DisableCtrButton(bool Close, bool Min, bool Max);
- void ShowScrollbar(BOOL Show);
- void GoTo(SHORT posX, SHORT posY);
- void ShowCur(bool CursorVisibility);
- void DisableSelection();
- SetConsoleOutputCP(437);
  The program uses some extended ASCII characters to make the game look prettier. We use this statement to set the output code page of the console to 437 in order to ensure that these characters will be displayed correctly on the console screen.