

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY



PROJECT REPORT THE MATCHING GAME

Class: Programming Techniques - 22CLC10

Supervisors:

Mr. Nguyen Thanh Phuong

Ms. Nguyen Ngoc Thao

Mr. Bui Huy Thong

Group of students:

Le Nguyen Gia Han - Student ID: 22127100

Tran Ngoc Uyen Nhi - Student ID: 22127313

Ho Chi Minh City, April 2023

Mục lục

1	Introduction	4
1.1	Pikachu Matching Game	4
1.2	Objectives	4
2	Overview	5
2.1	Game Tutorial	5
2.2	Program Format Explanation	5
2.3	Program Executing Instruction	8
2.4	Game's Rule	9
2.5	Game's Perfomance	9
3	Standard Features Explanation	10
3.1	Game Starting	10
3.2	I Matching	15
3.3	L Matching	15
3.4	Z Matching	15
3.5	U Matching	16
3.6	Read inputs from the keyboard	16
3.7	Game Finish Verification	17
4	Advanced Features Explanation	18
4.1	Color effects	18
4.2	Sound effects	18
4.3	Visual effects	18
4.4	Background	25
4.5	Move suggestion	26
4.6	Accounts and Score	27
4.7	Leaderboard	29
5	Extra Advanced Features Explanation	30
5.1	Stage difficult increase	30
6	Our Other Features	31
6.1	Login and Register	31
6.2	Menu Page	33
6.3	Print the game board information	34
6.4	Time played	34
6.5	Loading bar	34
6.6	Hide Cursor	35
7	Conclusion	36

ACKNOWLEDGEMENTS

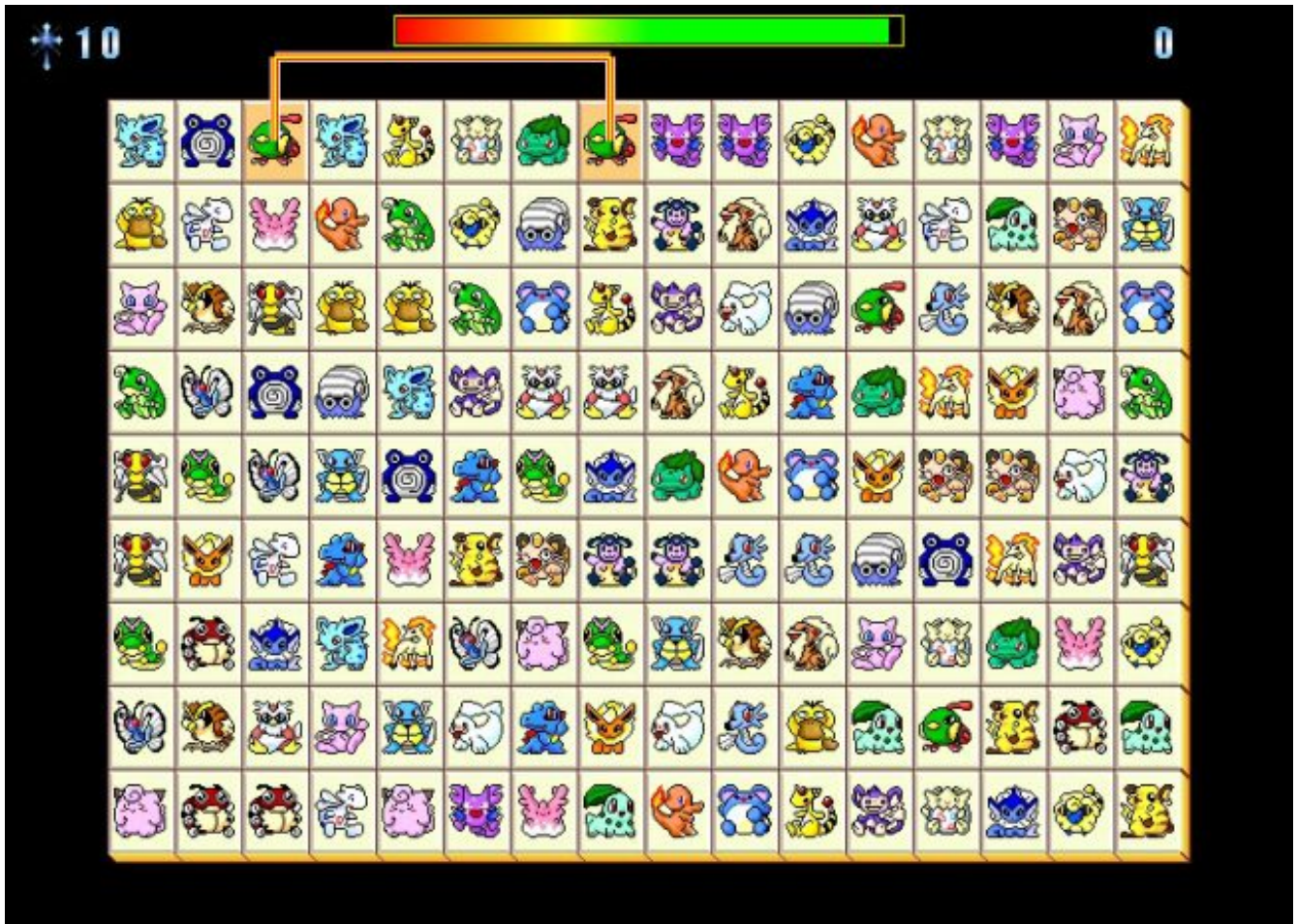
We would like to express our sincere thanks and deepest gratitude to Mr. Nguyen Thanh Phuong, Ms. Nguyen Ngoc Thao and Mr. Bui Huy Thong, instructors in the Department of Information Technology. Throughout the duration of this project, our team members have put in significant efforts in the process of searching and learning. However, we could not have achieved our goals without the valuable guidance and expertise sharing provided by our teachers. In the process of implementing the project, we acknowledge that there may be shortcomings in our work, therefore, we are looking forward to receiving the contribution of our teachers to help us learn and improve our skills for future projects. Once again, we would like to thank you very much, wish you a lot of health, success and always be dedicated teachers in the career of growing people.

GROUP OF STUDENTS

1 Introduction

1.1 Pikachu Matching Game

The classic Pikachu game, also known as the Pikachu matching game, is a popular puzzle game that originated in Japan. The objective of the Pikachu matching game is to clear all the tiles from the board by matching pairs of identical tiles that are decorated with pictures of various Pokemon characters. When two identical tiles are selected, they disappear and the player scores 1 points. The game continues until all the tiles have been cleared from the board.



Classic Pikachu game (from Google)

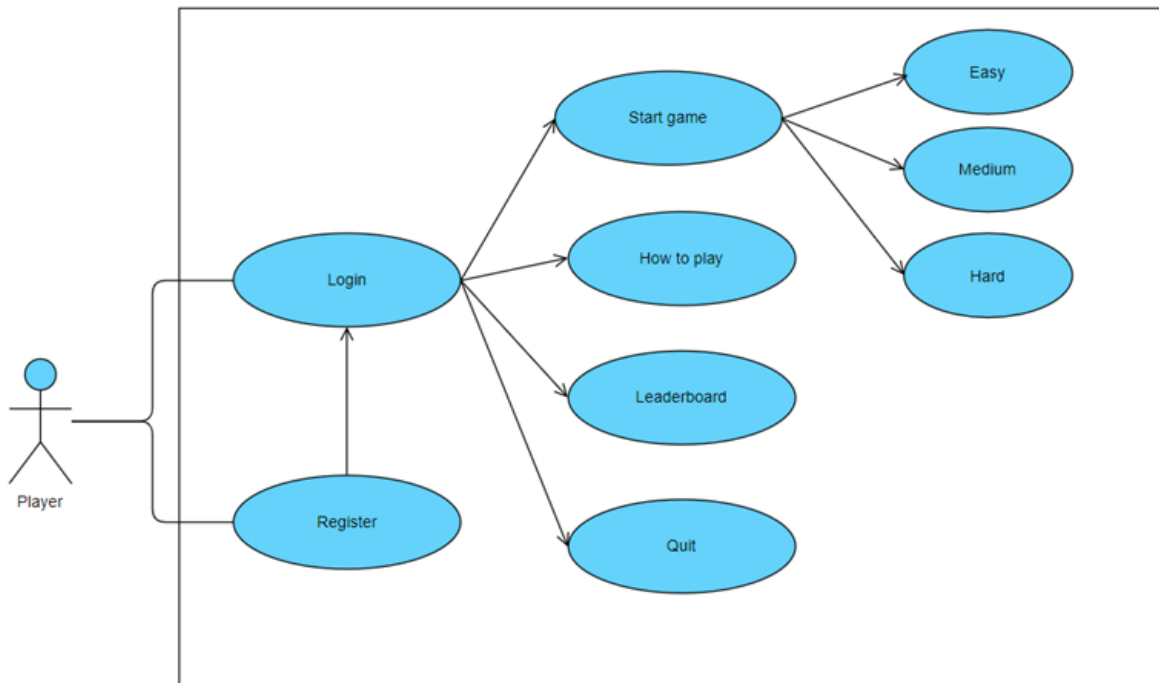
1.2 Objectives

In this project, we will develop a simplified version of the Matching Game by remaking the game with characters (instead of figures) using the programming language C++. The game must have an aesthetic interface and run stably, and it should include standard features such as tile shuffling, tile selection, and matching. In addition, we will incorporate

some advanced features such as sound effects, color effects, player scoring, play time tracking, and account creation and login. Our goal is to showcase how a game can be developed and enhanced to provide an engaging and enjoyable experience for players.

2 Overview

2.1 Game Tutorial



Use case diagram

2.2 Program Format Explanation

In our project, there are five files: Header.h, Menu.cpp, Board.cpp, Controll.cpp and Main.cpp files.

Header.h will contain all function prototypes.

Menu.cpp will include all functions used to display the selection screen (there are 2 types: Login selection and Menu game selection). Because the login part involves file storage for logging in, the functions for reading and writing binary files will also be included in this section:

```
void printLogin();  
void selectingLogin();  
  
Player* inputPlayerInfor(int& n);  
void login();
```

```

void registration();
void writeFile(Player* listPlayer);

void printWelcome();
void printMenu(Player &player);
void selectingMenu(Player* listPlayer, Player& player);

void loadingBar();
void printMode();
void playGame(Player& player);

void startGame(string MODE, Player &player);
void howToPlay();
void leaderboard(Player* listPlayer);
void exitGame();

```

Source.cpp plays the role of the core so that the game can operate, it includes functions used to create a matrix for the board and functions to check if 2 Cubes are connected and how they are connected:

```

void Shuffle(char** Letters, int SIZE);
void getLetters(char**& Letters, int SIZE, int SIZE_BOARD);

bool checkY(int x1, int y1, int x2, int y2, char** Letters);
bool checkX(int x1, int y1, int x2, int y2, char** Letters);

bool checkI(Cube p1, Cube p2, char** Letters);
bool checkL(Cube p1, Cube p2, char** Letters);
bool checkZ(Cube p1, Cube p2, char** Letters);
bool checkU(Cube p1, Cube p2, char** Letters, int SIZE_BOARD);

void prepareSelected(Cube& p1, Cube& p2);
bool checkSolve(Cube p1, Cube p2, char** Letters, int SIZE);
bool canSolve(char** Letters, int SIZE, int SIZE_BOARD, Cube& help1,
Cube& help2);
void slideLeftDirection(char** Letters, int SIZE);
void slide4Directions(char** Letters, int SIZE);

void startGame(string MODE, Player &player);

```

Board.cpp will include functions used to draw the game board (display section). This includes functions to draw Cubes and functions to draw the background.

```

string** storeBackground(string MODE);

```

```

void drawBackground(int SIZE, string** bg, int i, int j);

void drawHeader();
void drawFooter();
void drawRowMid(char** Letters, int i, int j);
void drawRowMidChosen(char** Letters, int i, int j);
void printSpace();

void drawHorizontalLine(int x1, int x2);
void drawVerticalLine(int x1, int y1, int x2, int y2);

void drawILine(Cube p1, Cube p2, char** Letters);
void drawLLine(Cube p1, Cube p2, char** Letters);
void drawZLine(Cube p1, Cube p2, char** Letters);
void drawULine(Cube p1, Cube p2, char** Letters, int SIZE_BOARD);

void drawBackgroundDeleted(string** bg, int i, int j);
void drawCube(char** Letters, string** bg, Cube previous);
void drawCubeChosen(char** Letters, string** bg, Cube selecting);
void drawCubeLock(char** Letters, Cube selected);
void drawCubeSuggest(char** Letters, Cube suggested);

void drawBoard(char** Letters, string** bg, int SIZE, int SIZE_BOARD);
void printInterface(Player& player, int score);

bool checkMatch(Cube p1, Cube p2, char** Letters, string MODE, int SIZE,
int SIZE_BOARD, int &score, int &time);
Cube Selecting(char** Letters, string** bg, int SIZE, Cube selecting,
Cube lockCube, Cube help1, Cube help2, int &score);
void Matching(char** Letters, string** bg, string MODE, int SIZE,
int SIZE_BOARD, Player& player);

```

Control.cpp includes game settings functions and controls functions to help players interact with the game in a simple way.

```

void setConsole();
void hideCursor();

void setColor(int background_color, int text_color);

int inputKeyboard();
void moveCursorToXY(int x, int y);

int calCubeWidth(int PosX);

```



```

int calCubeHeight(int PosY);

void movetoMiddleTopCube(Cube p);
void movetoMiddleBotCube(Cube p);
void movetoMiddleRightCube(Cube p);
void movetoMiddleLeftCube(Cube p);

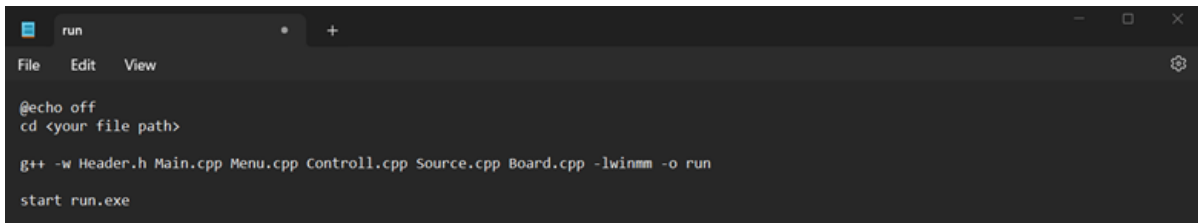
```

Main.cpp function will be called in this file to start the game.

2.3 Program Executing Instruction

Player can follow the instruction below to start the game (the tutorial for Windows 11):

- **Step 1:** Download and extract the folder "22127100-22127313".
- **Step 2:** Open the folder then right click the file 'run.bat'. Chose show more option and then Edit.
- **Step 3:** Copy the address of the folder named 'MatchingGameproject' and paste into line path <your filepath here>



```

run
File Edit View
@echo off
cd <your file path>
g++ -w Header.h Main.cpp Menu.cpp Controll.cpp Source.cpp Board.cpp -lwinmm -o run
start run.exe

```

Pasting folder address tutorial



```

run
File Edit View
@echo off
cd D:\22127100_22127313\MatchingGameproject
g++ -w Header.h Main.cpp Menu.cpp Controll.cpp Source.cpp Board.cpp -lwinmm -o run
start run.exe

```

Example of a legally folder address pasting

- **Step 4:** Run file 'run.bat', the game should be started then.



Game starting screen

2.4 Game's Rule

The Matching Game involves a grid of tiles, each displaying a unique letter. There are paths called I Matching, U Matching, L Matching, and Z Matching. The player has to connect pairs of squares so that they match the corresponding path.

Player's duty is to match all pairs of tiles with the same letter. If a pair doesn't match, you must make another attempt. Also selecting the same tiles is not allowed.

The game will end when all tiles have been successfully matched.

2.5 Game's Performance

Watch demonstration of the game at link: <https://youtu.be/f4zLD1WPaUE>

3 Standard Features Explanation

3.1 Game Starting

```
void getLetters(char**& Letters, int SIZE, int SIZE_BOARD);  
void Shuffle(char** Letters, int SIZE);  
bool canSolve(char** Letters, int SIZE, int SIZE_BOARD, Cube& help1,  
Cube& help2);
```

Before the player can start playing game, the program must create an matrix. The `getLetters` and `Shuffle` functions is used for this requirements.

getLetters(): This function will be responsible for allocating a matrix depending on the `SIZE` (which depends on the `MODE` the player chose). The algorithm used for this function is that it will sequentially assign the same pairs of elements to this matrix (since the task of the game is to connect the same cubes corresponding to the same letter, we must ensure that when a character exists, it must have a twin brother accompanying it). We will use the assignment statement for each pair of elements so as they will both random, to generate the same letter.

```
Letters[a][b] = Letters[a][b + 1] = rand() % 26 + 'A';
```

Shuffle(): Because if we only generate adjacent identical letters, the game will not be interesting. We need to shuffle their positions, and the `Shuffle` function is created for this purpose. It will use 2 for-loops to iterate through each element of the matrix. While iterating through each element, it will randomly select a row and column position, then swap these two elements. This process will move each element in the array to a random position.

canSolve(): After create a matrix, there are some opportunity for the program to create an unsovable matrix. This matrix has no use for the game because the player will never find a pair of Cube to match. So that the *canSolve* function was created for checking if the matrix can be solve or not, if it return false, it call again the *getLetters* function to create a new matrix (or it just simply use the *Shuffle* function to change the position until there some pair of Cube can be match).

This function use 2 pair of for-loop, the first pair for-loop is use to access through the elements of the matrix, the second one will also to access through the elements of the matrix but it use to checkMatch the elements accessed in the first pair for-loop to the remaining elements (it assign the position of the elements to 2 Cubes help1, help2 and send back the 2 Cube when calling function for feature Move Suggestion).

+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
		A					B					E		
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
		B					A					F		
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
		J					K					H		
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+
		K					J					N		
+	-	-	-	+	+	-	-	-	+	+	-	-	-	+

Example of Unsolvable matrix

```

void Shuffle(char** Letters, int SIZE);
void playGame(Player& player);
void startGame(string MODE, Player &player);
void getLetters(char**& Letters, int SIZE, int SIZE_BOARD);
bool checkSolve(Cube p1, Cube p2, char** Letters, int SIZE);
void Matching(char** Letters, string** bg, string MODE, int SIZE,
int SIZE_BOARD, Player& player);
bool checkMatch(Cube p1, Cube p2, char** Letters, string MODE, int SIZE,
int SIZE_BOARD, int &score, int &time);
Cube Selecting(char** Letters, string** bg, int SIZE, Cube selecting,
Cube lockCube, Cube help1, Cube help2, int &score);

```

To start the game, the player must chose Playgame in the Menu board. The *playGame()* function will be call and then ask the Mode the player want to play with. After chosing, it call the *startGame()* function and send the MODE been chosen to it, the Game then will be started.

playGame(): Draw the level selection board and call the *inputKeyboard* function for the player to choose the mode they want to play, then pass this mode as a string to the *startGame* function.

The reason for separating the functions playGame and startGame is that we have created separate functions for each selection in the Menu. Since playGame is also a selection, it needs to create a separate functions and needs to fulfill the task of allowing the player to choose a game mode. This feature should be separated from the startGame function to make the code look better.

startGame(): This function is responsible for allocating the char** Letters (the game board) and calling storeBackground() to store the background file in the string** bg. The size of the board and which background will be stored will depend on the mode chosen by the player. The game will officially begin when the Matching function is called. After the game, as well as the Matching function, ends, the startGame function will deallocating the 2D arrays.

Matching(): This function is used as the Main of the game, which helps the game to operate.

At the beginning of the game, the number of cubes will be SIZE*SIZE, corresponding to the number of cells in the playable area. The game will end when all the cubes are connected, and the number of cubes becomes 0. Therefore, the main part of this code is a While loop that continuously checks whether there are any remaining cubes in the game.

```

do {
    getLetters(Letters, SIZE, SIZE_BOARD);
} while (!canSolve(Letters, SIZE, SIZE_BOARD, help1, help2));

```

Before playing (before the while(cube!=0) loop begins), the program will call the *getLetters*

function to create a playable game board (to create a matrix has at least one pair of cubes must be connectable (the matrix is able to solve)).

The while loop (while cube != 0) will then begin, and it will call the *Selecting* function twice to allow the player to choose any two cubes to connect.

```
do {
    system("cls");
    drawBoard(Letters, bg, SIZE, SIZE_BOARD);
    printInterface(player, score);
    firstCube = Selecting(Letters, bg, SIZE, secondCube, lockCube, help1,
        help2, score);
    drawCubeLock(Letters, firstCube);
    secondCube = Selecting(Letters, bg, SIZE, firstCube, firstCube, help1,
        help2, score);
    drawCubeLock(Letters, secondCube);
    prepareSelected(firstCube, secondCube);
} while (!checkMatch(firstCube, secondCube, Letters, MODE, SIZE, SIZE_BOARD,
    score, time));
```

Before *Selecting* 2 Cubes, the *drawBoard* and *drawInterface* functions will be called to display the game board for the player to observe - interact with, and notify the player about their information, score, and game messages. This function will return two Cubes (Cube store the position of the selected elements in the char** Letters array). Then the *checkMatch* function is used to check if the two cubes can actually connect. If not, the player will have to choose again until *checkMatch* returns true. The do-while(*checkMatch*) loop is used for selecting two cubes because selecting two cubes must be done at least once when the while-loop (while cube != 0) begins. If a while-loop is used instead, the player will not be able to select any cubes for each pair of deleted cubes, because the *checkMatch* function of the while(*checkMatch*) loop can return any true or false value since no cubes have been selected to *checkMatch*.

While selecting, *drawCubeLock* will be called to set the Color for the Cube the player has chosen to reminds the player which Cube they has chosen. At first there is no Locked Cube so that we intilize it with the 0,0 – the position of non-playable area so that the *Selecting* function won't change the Color of this Cube.

The *Selecting* function also requires an optional Cube to be the first one to change color (note that this cube is being pointed to), and this cube will be changed by the player's control. Therefore, while selecting the first cube, we will first send a cube to be the one being pointed to. This cube will be decided to be the cube at position 1,1 (the beginning of the matrix). After *Selecting* the first cube, this cube will be passed to the second selecting function as if the player will start selecting the second cube from the position of this cube.

checkMatch(): This function receives two cubes, which correspond to the positions of the elements that the player has chosen, and checks whether they can be connected using four functions: *Check I*, *Check L*, *Check U*, and *Check Z*. The function *checkMatch*

will return true if one of the four conditions returns true and simultaneously call the corresponding drawLine function for one of the four shape check functions, then increase the player's score for each correct Match.

Selecting(): This function will use the *inputKeyboard* function to navigate through the matrix by receiving corresponding inputs (WASD keys, Arrow keys and Enter keys), increasing or decreasing the position of the starting cell, and returning the position of the chosen cell after the player presses Enter.

Up - W keys: When this key is pressed, it seems like the player wants to move up one cell from the current cell. After pressed, the row of the current cell (Cube.y) will be decreased by 1 and become and it now become the current cell. The *drawCube* and *drawCubeChosen* functions will be used to adjust the color of the cell. Set the blue color for the current cell and the default white color for the old cell so that the player can identify which cell is the current one.

The similar code is used for the others keys. Down - S keys: Increase the row of the current cell (Cube.y) by 1. Right - D keys: Increase the column of the current cell (Cube.x) by 1. Left - A keys: Decrease the row of the current cell (Cube.y) by 1.

When the H-keys is pressed, it tells that the player would have to get some move suggestions, it then draw the Color for Cube help1 and help2 (which intialized in the canSolve function) for the player to know which cube can be connected.

Before checking the type of matching between two Cubes, I created two functions, checkY and checkX, to verify the Horizontal and Vertical lines respectively. These functions will be used to check the matching shape. Assume that the empty Cube is the Cube which char** Letters[Cube.y][Cube.x] == ' ' (space characters) and the pathway must be fill with the character ' ' (char** Letters[y][x] == ' '). If the line has a alphabet character, It will not be considered as a pathway.

```
bool checkY(int x1, int y1, int x2, int y2, char** Letters);
bool checkX(int x1, int y1, int x2, int y2, char** Letters);
```

checkY(): This function takes the coordinates of the "row line" that you want to check as parameters. Its duty is to check if there is a pathway (no obstacles) between two points x1 and x2 (corresponding to column x1 and x2). Since its task is to check in the same row, if you pass parameters such that y1 is different from y2 (not the same row), it will return false. When y1 = y2, it will make the for-loop run from the smallest x position to the largest x position to check if the straight line between them forms a pathway or not. Note that it only checks the pathway between the two indexed columns (x), not the start and end positions. Therefore, depending on the purpose of using this function, we may need to add a condition to check for any obstacles at the x1 and x2 positions.

checkX(): This function operates similarly to the checkY() function, but its purpose is to check if there is a pathway between two points y1 and y2 (corresponding to row y1 and y2) if they have the same x (the same column line).

3.2 I Matching

```
bool checkI(Cube p1, Cube p2, char** Letters);
```

This function will return true if there is an I-shaped match between 2 cubes. First, it checks whether the 2 cubes are in the same row or the same column, and if not, the function immediately returns false (to form an I shape Cubes must have the same column or same row). If they are in the same row or column, the function then uses checkY() and checkX() to determine whether the pathway between the 2 cubes forms an I shape. If either checkY() or checkX() returns true, then an I-shaped match is formed. If both return false, the program will continue to check for other shapes.

3.3 L Matching

```
bool checkL(Cube p1, Cube p2, char** Letters);
```

It first checks if there is a vertical line connecting the two cubes using checkY() and a horizontal line using checkX(). If both of these lines exist and the bend between 2 lines have no obstacle (form an empty Cube), then an L-shaped match is formed. Similarly, the next condition it checks for a horizontal line connecting the two cubes using checkX() and a vertical line using checkY(). If both of these lines exist and there is no obstacle between to line, it then an L-shaped match is also formed.

If either of these conditions are correct, the function returns true. Otherwise, it returns false and the program would start checking for the next shape.

3.4 Z Matching

```
bool checkZ(Cube p1, Cube p2, char** Letters);
```

There are two types of Z-shaped: One is formed by 2 Vertical Lines and 1 Horizontal Line, and the other is formed by 2 Horizontal Lines and 1 Vertical Line.

For the first case, the function checks from the smallest y row to the largest y row (pMinY.y and pMaxY.y). A condition is added to determine which Cube will be the pMinY and the pMaxY, and assigns them to these two variables. Then the function starts a for-loop from the pMinY + 1 to pMaxY - 1 to find if there is a vertical line that satisfies the conditions to form a Z-shape between the two Cubes. To meet the above condition, the vertical line under consideration must be a pathway itself and the two horizontal lines connecting the two Cubes to the beginning and end positions of the vertical line must also be pathways. Additionally, there should be no obstacles at the beginning and end positions of the vertical line. When these conditions are met, the function will return true, indicating that the two Cubes have formed a Z-shape.

The similar code is used for the second case. In this case we don't need to compare which is the smaller x and the larger x, the *prepareSelected* function has done it. So let's assume that the pMinX = p1 and the pMaxY = p2. It use a for-loop to check if there is a horizontal line that satisfies the conditions to make sure that the Cube p1 and p2 form a Z-shaped.

It then check if the 2 horizontal line satisfies the conditions, and if there are any obstacles at the beginning and end of the Horizontal line or not.

If none of these two case return true, the 2 Cubes didn't form a Z-shaped.

3.5 U Matching

```
bool checkU(Cube p1, Cube p2, char** Letters, int SIZE_BOARD);
```

There are 4 types of U-shaped: Inverted U, Up-right U, Right-facing U, Left-facing U. Both Right-facing U, Left-facing U are formed by 2 Horizontal line and 1 Vertical line, while Inverted U, Up-right U are formed by 2 Vertical line and 1 Horizontal line.

So that I will use the similarly code of the checkZ function for this function. For the Inverted U, Up-right U we still consider which is the smaller or larger y and assigns them to the variable pMinY and pMaxY. They use the same conditions as the checkZ function but the for-loop is a little different.

While other functions only consider the playable area (SIZE*SIZE) in the for-loop, the drawULine function uses a for-loop to iterate over cells in both the playable and non-playable areas (SIZE + 2)*(SIZE + 2). So that for the Inverted U, the for-loop should be start from pMinY.y - 1 row and end at 0 row (The playable area has the indices start from 1,1). Up-right U for-loop will start from pMinY.y + 1 row and end at SIZE_BOARD - 1 row (The playable area has the indices end at SIZE_BOARD -2, SIZE_BOARD - 2). The similarly code will be used for both Right-facing U, Left-facing U too.

The function will return true if the Cube form an U-shaped at every code for types of U, otherwise if there is no return true be called, it will return false.

3.6 Read inputs from the keyboard

The function "inputKeyboard" uses the "_getch" function from the "conio.h" library to read a character from the keyboard buffer.

This declares an integer variable named c and initializes it to the value returned by the _getch() function. This function waits for a keypress and returns the ASCII code of the pressed key.

The reason for having two _getch() calls here is that certain special keys, such as the arrow keys, are represented by a sequence of two characters when they are pressed. The first character is either 0 or 224, indicating that a special key has been pressed, and the second character specifies which key it is (for example, 72 for the up arrow). So, when _getch() is called the first time, it reads the first character of the sequence. If it is 0 or 224, then we need to read the second character to determine which key was pressed.

If an arrow key is detected, the function returns a value between 1 and 4, representing UP, LEFT, RIGHT, and DOWN respectively. If the character is not an arrow key, the function checks if it matches any of the predefined keys. If the character matches any of the predefined keys, the function returns a corresponding integer value. For example, if

the character is 'W' or 'w', the function returns 1 to represent the UP key. Similarly, if the character is 'A' or 'a', the function returns 2 to represent the LEFT key, and so on. If the character does not match any of the predefined keys, the function returns 0, indicating that no valid input was detected. This could happen, for example, if the user presses an unsupported key.

The function can detect the following keys: Arrow keys (UP, LEFT, RIGHT, DOWN), Enter key, Help key (H), W, A, S, D keys.

3.7 Game Finish Verification

The game will be finished if the Cube remaining is 0 (All Cube are matched) as the description I mentioned in Matching function. After the game finished (which means the *playGame* function has ended its job). The `writeFile` function is called to store the performance of the player in to the binary file and use it for leaderboard.

4 Advanced Features Explanation

4.1 Color effects

The function `setColor()` is used to change the color of the text and background in the console window. The function takes two integer parameters: background and text, which represent the desired background and text colors, respectively.

Firstly, we define a set of constants for specific color codes in **header.h** file using **define** statements. These constants represent the 16 different color combinations that are available in the console window.

Inside the `setColor()` function, it first calls the function `GetStdHandle` to retrieve a handle to the console output. The console output handle is an identifier that can be used to manipulate the console window.

Then, the function calculates a color attribute value based on the selected background and text colors. The color attribute is a single value that specifies both the background and text colors using a bit pattern. The background color is multiplied by 16 to shift its bits to the left and make room for the text color. The two values are then added together to create the final color attribute. Finally, the function calls the function `SetConsoleTextAttribute(hConsole, colorAttribute);` to set the console window's text and background colors. The functions `GetStdHandle` and `SetConsoleTextAttribute` are declared in the **windows.h** library.

4.2 Sound effects

We use the **mmsystem.h** library and link **winmm** to the compiler in order to have function `PlaySound()`.

The function uses the **PlaySound** function to play the sound file. The first parameter of `PlaySound` is the name of the sound file to play, the second parameter is a handle to the sound device to use, which is specified as `NULL` to use the default sound device. The third parameter is a flag that specifies how the sound should be played. The `SND_ASYNC` flag specifies that the sound is played asynchronously and `PlaySound` returns immediately after beginning the sound.

For instance, every time the users press an arrow key, `PlaySound(TEXT("move.wav"), NULL, SND_ASYNC)` is called to play the move sound.

4.3 Visual effects

For this effect, we will add color to the Cube and draw a connecting path between two matching cubes. First of all, we created 2 functions for moving the cursor to the position of the Cube on the board. `int calCubeWidth(int PosX); int calCubeHeight(int PosY);`

calCubeWidth(): This function will return the x-coordinate of the cube by multiplying its column position by 7 (since each cube has a width of 7).

calCubeHeight(): This function will return the y-coordinate of the cube by multiplying its row position by 3 (since each cube has a height of 3).

There are four functions created for the color of the cube:

```
void drawCube(char** Letters, string** bg, Cube previous);  
void drawCubeChosen(char** Letters, string** bg, Cube selecting);  
void drawCubeLock(char** Letters, Cube selected);  
void drawCubeSuggest(char** Letters, Cube suggested);
```

In each of the four functions above, the *calCubeWidth()* and *calCubeHeight()* functions are used to return the coordinate of the middle left of the cube. The cursor is then moved to this position and the color is changed based on the function being called.

drawCube(): The function takes two 2D arrays 'Letters' and 'bg' (for the *drawBackgroundDeleted* function), and a Cube object "previous" as its arguments. This function is used to change back the original color (White) of the cube if the cube is not deleted yet (that is why it is called "previous" Cube). If the cube is deleted, it will change back the original color of the background of that cell on the board.

drawCubeChosen(): This function has the same usage as the *drawCube* function, but instead of changing back the color of the previous Cube (or background), it changes the color of the current Cube (or background) that is being selected (that's why it is called the "selecting" Cube) to Blue.

drawCubeLock(): This function takes a 2D character array Letters and a Cube object "selected" as its arguments. It is used to change the color of the cube being locked (which means being chosen by the player) to Red (that's why it is called the "selected" Cube).

drawCubeSuggest(): This function takes a 2D character array Letters and a Cube object "suggested" as its arguments. It is used to change the color of the cube being suggested by the program ("Help1" and "Help2" Cube) to Green (that's why it is called the "suggested" Cube). Since there are two cubes for move suggesting, it needs to be called twice at a time.

There are 2 types of function created for the path way of two Cubes:

The first type is used to move the cursor to the middle of the cube to ensure that the pathway starts from the center of the cube, resulting in a better-looking pathway.

```
void movetoMiddleTopCube(Cube p);  
void movetoMiddleBotCube(Cube p);  
void movetoMiddleRightCube(Cube p);  
void movetoMiddleLeftCube(Cube p);
```

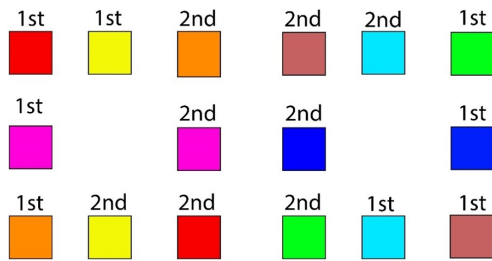
In each of the four functions above, the *calCubeWidth()* and *calCubeHeight()* functions are used to return the coordinates of the middle of the cube based on the function being

called. For example, *movetoMiddleTopCube()* returns the coordinates of the middle top of the cube and so on.

This function was created while writing the draw pathway functions. It is used to swap the values of two cubes such that Cube p1 (first Cube chosen) always has a smaller x value than Cube p2 (second Cube chosen). If both cubes have the same x value, the cube with a smaller y value becomes Cube p1. If we don't use this function before drawing the pathway between two cubes, there would be difficulties. For example, when drawing a **Horizontal I-line**, we need to know which cube is on the left (has smaller x) so we can start drawing from it, as the cursor will move from left to right naturally with the characters output by programmers. If we start drawing the pathway from the cube on the right, we would have to move the cursor to the left every time we draw a character, which is very time-consuming. Therefore, knowing which cube to start from is crucial. The same goes for drawing a **Vertical I-line**, if we assume that the first Cube is the above Cube (smaller y), we can design only one code to move and draw the pathway from the first Cube to the second Cube by drawing from the top to the bottom instead of writing an additional code to draw in the opposite direction.

Once a gain, by using the function, we don't have to consider which Cube will be on the left or which Cube will be on the above to start drawing, we will know that the firstCube pass to the drawLine function will be the Cube on the left (or on the top), so that we just design the function to draw the Line from this Cube only.

This function will be called before drawing the pathway between two Cubes.



Before using prepareSelected function (We have to consider 8 cases)



After using prepareSelected function (We only consider 4 cases)

The second type of function is used for drawing the pathway between two cubes.

```
void drawHorizontalLine(int x1, int x2);
void drawVerticalLine(int x1, int y1, int x2, int y2);
```

These two functions is used to draw Horizontal and Vertical Line, the reason why we pass specific x and y (represented for the column and row) instead of passing Cube structures and assuming that Cube.x represents the column and Cube.y represents the row is because this function is drawn only one horizontal lines and vertical line, but there may be one or

two horizontal lines and vertical lines based on the matching shape between two Cubes, passing specific rows and columns allows more flexibility in drawing the lines.

drawHorizontalLine(): This function takes two parameters representing the column indices. It compares the two indices to determine which one is greater, calculates the subtraction between them, and begins drawing the path from the smaller column index. The distance between 2 columns, $1 \leq \text{length} = x_2 - x_1$ (x_2 different from x_1). If the subtraction is equal 1, it means that there is no space between the two columns, so we don't need to draw anything. That's why the for-loop, which is used to repeat drawing the pathway for each column, starts from 1 and ends at $\text{length} - 2$ (where 2 is the number representing the two Cubes at the two ends of the distance).

drawVerticalLine(): This function takes four parameters representing the column and row indices. It compares the two row indices to determine which one is greater, calculates the subtraction between them, and begins drawing the path from the smaller row index. The reason why we need the index of the column is that when we use the `cout` statement to output something, the cursor will appear at the right of the last character. However, drawing vertically means drawing under each character in the same vertical line. Therefore, we need the column index for the *moveCursorToXY()* function. The for loops start from 1 and end at $\text{length} - 2$ for the same reasons as in the *drawHorizontalLine()* function

```
void drawILine(Cube p1, Cube p2, char** Letters);
void drawLLine(Cube p1, Cube p2, char** Letters);
void drawZLine(Cube p1, Cube p2, char** Letters);
void drawULine(Cube p1, Cube p2, char** Letters, int SIZE_BOARD);
```

Since the pathway should be drawn in red color for the background, and white color for the text, the *setColor* function is called for each of the functions above to achieve this. Once the pathway has been drawn, the *setColor* function is called again to change back the origin color of the game. This is important because the pathway is only temporary and should not permanently alter the colors of the game.

Before going into details, let's assume that the index of column and row of the first Cube chosen is $p1.x$ and $p1.y$, and the second Cube chosen is $p2.x$ and $p2.y$.

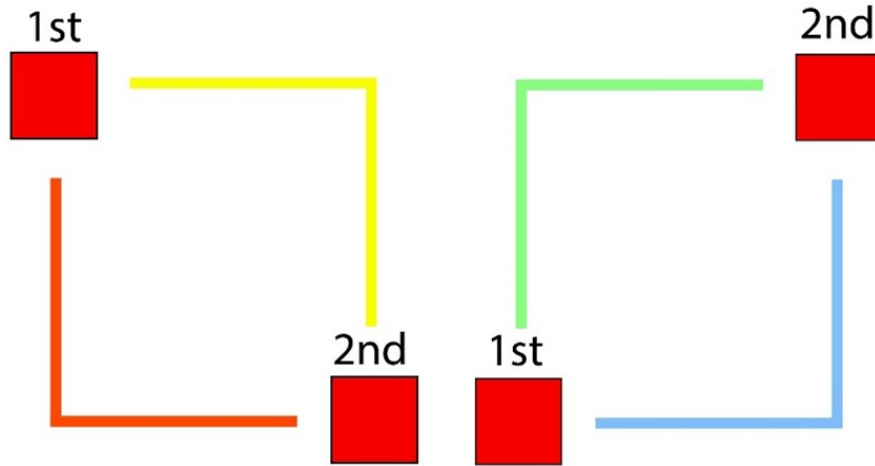
drawILine(): The function takes a 2D character array called "Letters", and two Cubes "p1" and "p2". It first checks whether the two Cubes are on the same row or column by comparing their "y" and "x" values:

If the two Cubes have the same y value (are on the same row), the program moves the cursor to the middle-right of the first Cube and middle-left of the second Cube by using the *movetoMiddleRightCube* and *movetoMiddleLeftCube* function and put a character '=' for each Cube to mark the end of the line. However after called the *movetoMiddleRightCube* function, the *drawHorizontalLine* will immediately call because the cursor is now sit in the right position to draw the pathway.

If the two cubes are on the same column (same x value), we will use *movetoMiddleBotCube* and *movetoMiddleTopCube* function to put characters '=' at the bottom of the

first Cube and the top of the second Cube. It then calls the `drawVerticalLine` function to draw a vertical line between the two cubes to represent the Vertical I line.

drawLLine():The function use the `checkX` and `checkY` function to consider which the L line shape would be.



L Line

There are 2 main cases:

1. $p1.y < p2.y$ (The left one in figure)

a) The L-shaped line connects the first and second cubes by first forming a horizontal line and then a vertical line. The `drawLLine` function first checks whether a horizontal line (`checkY`) can be drawn in the $p1.x$ row between the $p1.x$ column and the $p2.x$ column. It also check whether a vertical line (`checkX`) can be drawn in the $p2.x$ column between the $p1.y$ and $p2.y$ rows. Because the `checkX` and `checkY` funtions only check the space between the two positions to see if there is any empty cell, rather than checking if there is any empty cell at the exact positions themselves, so it neccesary to add one more condition to check is there an empty cell right the corner (It can shape the Z Line if there are not an empty cell right there). If these codition are true, function start drawing.

The function starts using `movetoMiddleRightCube` for $p1$ Cube and put characters ‘=—’ to mark the start of the line (The ‘—’ is adding next to the ‘=’ char because we want it to set the cursor in the center after call `drawHorizontalLine` function). The `drawHorizontalLine` function is called imediately after that (because the cursor are at the right position after putting ‘=—’ char) to draw a line between the two columns in the same row.

After drawing the horizontal line, the function calculates the coordinates so that the ‘|’ character is positioned directly below the ‘+’ character, which is also the position above the start point for drawing the vertical line. This ‘|’ character is used to make the pathway look more “continuous”. Finally, it calls the `drawVerticalLine` function to draw the verti-

cal line between the two cubes and prints another equals sign '=' to mark the end of the line.

b) The L line would shape like the orange line, which means the first Cube and second Cube first connect together by a Vertical line and then a Horizontal line. The function first checks whether a vertical line (*checkX*) can be drawn in the p1.x column between the p1.y row and the p2.y row, and whether a horizontal line (*checkY*) can be drawn in the p2.y row between the p1.x and p2.x columns. And of course there is a condition to check whether the corner is an empty with the same reasons mentioned in the case above.

If the line follows this path, the function starts using *movetoMiddleBotCube* for p1 Cube and put characters '=' to mark the start of the vertical line and immediate draw the Vertical line in column p1.x from p2.y to p1.y rows, and move the Cursor right on top the Vertical line to put '|' character.

Next we move the Cursor right above on the '|' character to draw the horizontal line and mark '=' for the end of the pathway.

2. p1.y > p2.y (The right one in figure)

This case has the same description as the first one, the difference is that we have to adjust the different cursor coordinates for moving it and use another *movetoMiddle* function for each Cube in order to position the '=' character correctly.

drawZLine(): In this function, we use 2 main conditions (*CheckY && CheckX && CheckY*) and (*CheckX && CheckY && CheckX*) to determine how to draw the Z Line. For the first condition (*CheckY && CheckX && CheckY*), this condition uses a for loop that iterates through each column starting from the position of the first Cube. Each column being considered will have a pair of pathway (from the position of the Column of 2 Cubes to the column being considered). It will then check if there is any column with its pair of pathway that is suitable for drawing the Z Line. The second condition (*CheckX && CheckY && CheckX*) is used to check if there is any row that is suitable for drawing the Z line, using a similar checking method. Every condition has to check whether if there are two empty cells (' ' Cube) at the bend of the line.

There are 2 main cases, every case will have 2 conditions (stand for 2 little case inside) mentioned above:

1. p1.y < p2.y

a) (*CheckX && CheckY && CheckX*) (The yellow line):

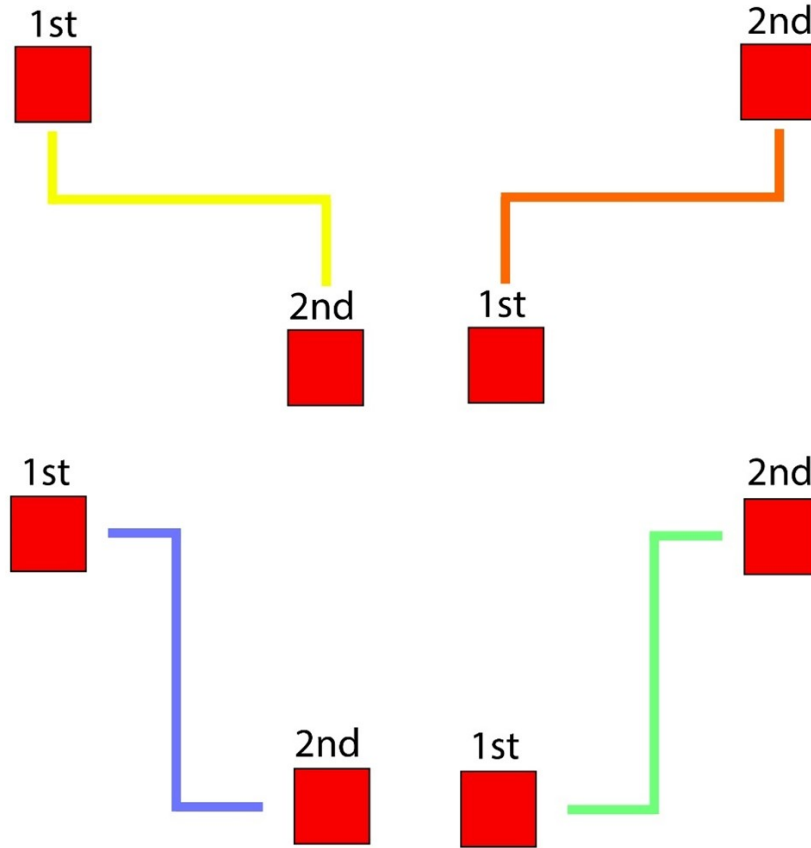
This part reuses a similar code as in section 1a) of the *drawLLine* function. However, after finishing drawing the L line, it will add '—+' to fill the pathway and draw another vertical line (which is equivalent to adding another '|' character). These following cases have the same method as this one.

b) (*CheckY && CheckX && CheckY*) (The blue line)

2. $p1.y > p2.y$

a) (CheckX && CheckY && CheckX) (The orange line)

b) (CheckY && CheckX && CheckY) (The green line)



Z Line

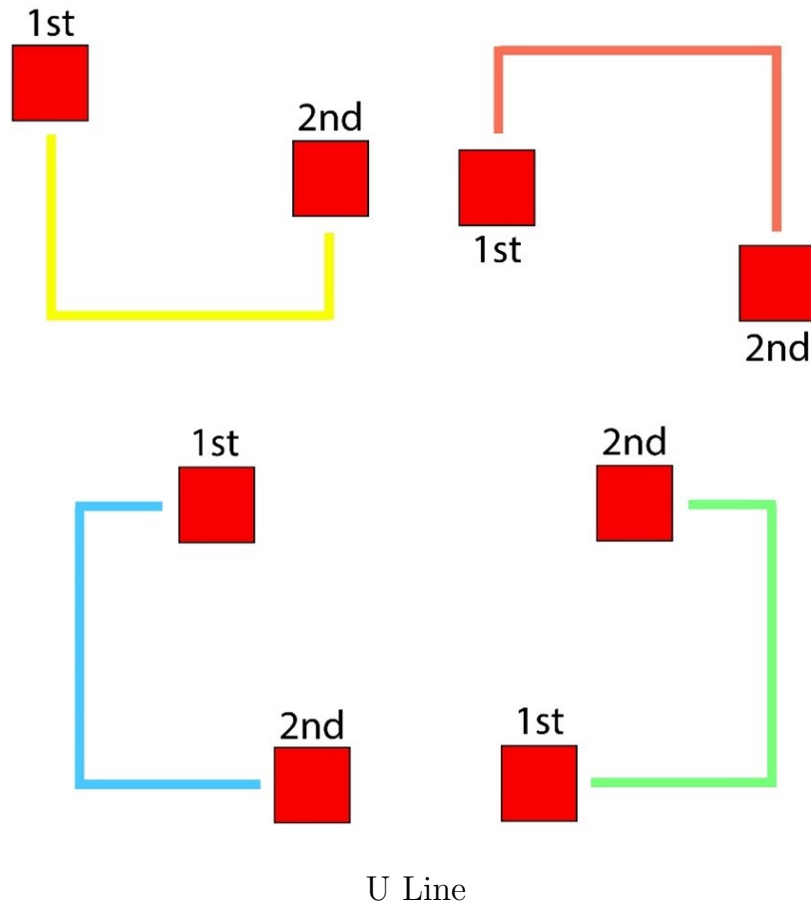
drawULine(): As a result of the *UMatching* function, the game is designed so that users can still consider a U-shaped path for 2 Cubes even when the path is not entirely within the playable area. This function is slightly different from others, as it requires an additional parameter, which is the `SIZE_BOARD`, to be used in the for-loop. While other functions only consider the playable area ($SIZE * SIZE$), the *drawULine* function uses a for-loop to iterate over cells in both the playable and non-playable areas $(SIZE + 2) * (SIZE + 2)$.

The method used for this function is the same as the one used for *drawZLine* (CheckX-CheckY conditions and check empty cells at the bends of the line), but it is a little more complex. I don't consider which Cube has the smaller or bigger index y row like the above functions for a little reason. If I slit the function into 2 case ($p1.y < p2.y$ or $p1.y > p2.y$) there will be 4 more case correspond to 4 U shape inside 2 main case and I have to use the CheckX-CheckY conditions for 4 of them, that means I need to use the conditions for 8 times. Slitting the function in 4 main case correspond to 4 U shape is also need 2 more case inside every of them, but I only need to use the CheckX-CheckY conditions for 4 times

only. Besides that it is better if we slit it into 4 main parts, the code will be easier to adjust.

There are 4 main cases:

1. Inverted U (The orange line):
2. Up-right U (The yellow line):
3. Right-facing U (The blue line):
4. Left-facing U (The green line):



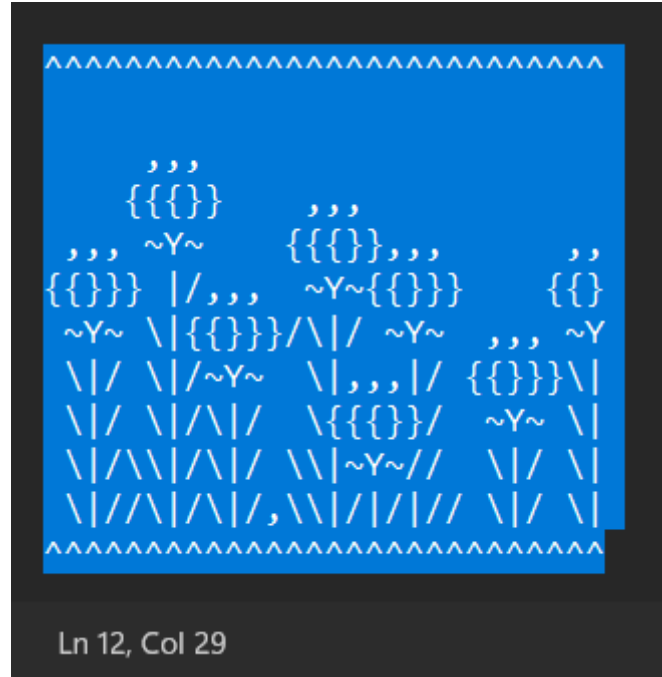
4.4 Background

This feature is used to draw the background on the board if there are empty cells (deleted Cube). We have 3 levels, so there are 3 backgrounds used. We use these three functions for this effect.

```
string** storeBackground}(string MODE);  
void drawBackground(int SIZE, string** bg, int i, int j);  
void drawBackgroundDeleted}(string** bg, int i, int j);
```

storeBackground(): After the player selects the game mode, this function will be called. The string "MODE" will determine which background to choose, and then memory

will be allocated for the string** "bg" to have $\langle \text{SIZE} \times 3 \rangle$ rows (the height of a cube is 3 rows) and $\langle \text{SIZE} \rangle$ columns (each column will have only 7 characters, as the width of a cube is 7). To ensure that the program reads the file correctly, it is important to make sure that each line of the background file has $\langle \text{SIZE} \times 7 + 1 \rangle$ columns and $\langle \text{SIZE} \times 3 \rangle$ rows.



Background for easy mode

drawBackground(): This function takes four parameters: the string** "bg" allocated in the function above, the SIZE of the playable area, and the indices that determine which cell of the board needs to be drawn with the background. Since the column and row indices of the playable area start from 1, but the background starts from index 0, when the background receives the indices and uses the *moveCursorToXY* function to move to the cell that needs to be drawn, the indices must be incremented by 1. This function will be called within the "drawBoard" function, which uses a for-loop to check whether each cell is empty and calls this function to draw the background for this cell.

drawBackgroundDeleted(): This function will change the original color for the cell that the player has just moved through (when the player moves across these cells, they will change to blue (drawCubeChosen function)).

4.5 Move suggestion

For this feature, I created this function:

```
bool canSolve(char** Letters, int SIZE, int SIZE_BOARD, Cube& help1,
Cube& help2);
```

(There is also the *drawCubeSuggestion* function which I mentioned above is used for this function. Also note that the algorithm used for this mentioned in the *Matching* function)

canSolve function will return the true value and send the position of two cubes for suggestion (Cube help1 and help2). If the function returns false, it will also send the positions for Cube help1 and help2, but it won't affect the game because the function will be called repeatedly (in the Matching function) until it returns true (send a correct suggestion Cube) while the player is playing the game.

4.6 Accounts and Score

We use binary file which the extension .dat to store the data of Players. There is 3 main function are used for this feature.

```
Player* inputPlayerInfor(int \&n);  
void login();  
void registration();  
void writeFile(Player* listPlayer);
```

inputPlayerInfor(): The function reads the data from a binary file name "records.dat", which contains the number of players in the list and their data. The Player* list is allocated and the data is stored in this list.

login(): The function will call the inputPlayerInfor function to store a list of Players and send back number of the players (int n).

For inputting the username and password, we have add a small feature name hidden password, this feature will turn the password into character '*' everytime we type a character. When we input a normal characters by normal cin statement (or cin.getline, getline etc.), we can easily use the backspace to delete the previous characters we typed. Using the hidden password is different from the input statement because it is output by the cout statement and it can not delete like normal, so we use the statement below to done this need.

```
password.erase(password.length() - 1);  
cout << "\b \b";
```

There is a condition check whether the BACKSPACE key ('_ - 8) is hit when typing the password. If yes, it will decrease the length of password by 1 (delete the last character in password) and then delete the '*' character using the next cout statement. The '_' character will move back the cursor right before the last '*' character and then put a space character ' ' to delete the '*', then move the cursor back right before the ' ' character to write the other characters for pass word.

Then it use for-loop start from the first elements to the last of list to check whether the username and password input by the player match any of the elements in the list or not. Since the username and password variables are declared as strings, they need to be compared with a char array (the data type used for the username and password in the Struct Player list). To do this, the c.str() function is used to convert the string to a c string, and the strcmp function is used for comparison.

registration(): This function will let the player input the username and password they want to use for their account, and then sets all game statistics of the player to zero (cause they haven't played yet).

The function first opens the file in ios::in | ios::out mode and checks if the file can be opened. If the file cannot be opened, the out mode will help create a new file and write (numbers of player) $n = 1$ to the beginning of the file (because after the function ends, the information of the first player has been saved, so the current number of players is 1). If the file can be opened, it means that an account has been created before. The function will move to the beginning of the file to read the number of existing players and write the new number of players (increased by 1), then move to the end of the file to add the information of the new registered player.

Afterward, the information of the new player will be write at the end of file.

writeFile(): This function will take the struct Player* list and save all the information of these players to a file. The ios::trunc mode is used to delete all old data in the file and write new data to update the players' information. The data written includes the number of players and the list of players passed in. This function should be called after the player finish playing the game to save the performance of the player's in the game.

```
struct Level {
    int score;
    int time; // seconds
};

struct Player {
    char username[20];
    char password[20];
    Level level[3]; // 3 level
};
```

The following two structures store information about the player, which define an account of player. The Player structure has two character array members to store the player's username and password, as well as a Level structure member to record the player's performance in the game. An array with three elements corresponds to the three levels of the game, and each level has a player's score and the time played.

The words "WELCOME", "MATCHING GAME" and "Hello Words" are turned into Ascii characters using the website <https://ascii.co.uk/text>.

4.7 Leaderboard

```
void leaderboard(Player* listPlayer);
```

After be chosen in the Menu, the function will be called and asking the user to choose between difficulty levels (using `inputKeyboard` function to navigate and chose a difficulty) and then displays the leaderboard for that level.

To show the leaderboard, the function must take an array of `Player` as input, which contains information about the players' performance in the game.

It first open the file to take the numbers of player and the make a copy of `Player* listPlayer` which name `Player* rank` for sorting purpose. This `Player* rank` will be used exclusively for sorting, so as not to shuffle the player list in `listPlayer`, which will be saved back to the file.

The leaderboard displays the rank, username, score, and time played of the players for each chosen game mode (difficulty level). That means in the EASY level, Player A might be the highest score player, but in the other level, the first rank maybe for someone else. The players are ranked based on their score and time played, with higher scores and shorter times played being ranked higher. The leaderboard is sorted in descending order using interchange sort. The maximum players appear on the board is 5.

5 Extra Advanced Features Explanation

5.1 Stage difficult increase

For this feature, I created 2 levels MEDIUM and HARD. In the MEDIUM level, the board game will be sliding to the Left, and in the HARD level, the board game will be sliding all four directions (Left, Down, Right, Up).

When looking at the Example of left-sliding section in the Project instruction file, I immediately thought of "sorting" it so that the empty Cube part would move to one side and the non-empty Cube part would move to the other side without disrupting the order of occurrences of the non-empty Cubes. Therefore, I used a Bubble Sort algorithm to do this. Since Bubble Sort can help push the empty Cubes one by one to the end of the row while still preserving the positions of the Cubes, I thought this was a suitable choice.

Then I created 2 Sliding functions for the 2 levels as mentioned above:

```
void slideLeftDirection(char** Letters, int SIZE);  
void slide4Directions(char** Letters, int SIZE);
```

slideLeftDirection(): This function is used for the Medium level, and it simply uses a large outer loop to go through each row in the board. For each row it goes through, it uses two inner loop goes through each element in the row from the second element to the end. If the inner loop encounters an empty character in a certain position of the row, it swaps that character with the character in the next position. This way, each empty character is moved towards the end of the row until it reaches the last position.

```
for (int i = 1; i <= SIZE; i++)  
    for (int m = SIZE; m >= 1; m--)  
        for (int n = 2; n <= m; n++)  
            if (Letters[i][n - 1] == ' ' )  
                swap(Letters[i][n - 1], Letters[i][n]);
```

slide4Directions(): This function use the same method as the above. In this function, I divide the board into two halves, the left half and the right half. The left half will be the left-sliding while the right half will be right-sliding. I also divide the board into another two halves, the top half and the bottom half. The top half will be up-sliding while the bottom half will be the down-sliding.

For left-sliding, I used the same code as the slideLeft function, but the first inner loop will be start from $SIZE/2$, cause now I only need to slit Left one side (Left half). The code for Top-slide is similarly but now the i value (the value of the first for-loop) will be used for the index of the column.

For right-sliding, there is a little difference from left-sliding. The second loop iterates through each element in the right half of the row, starting from the middle index $SIZE/2 + 1$ (because $SIZE/2$ returns a value that belongs to the left half) to the end index $SIZE$.

The innermost loop iterates from the current index n to the middle index m in descending order. In innermost loop, the code checks if the character in the current index $n+1$ is empty or not. If it is empty, it swaps the current character n with the next character $n+1$ in the array.

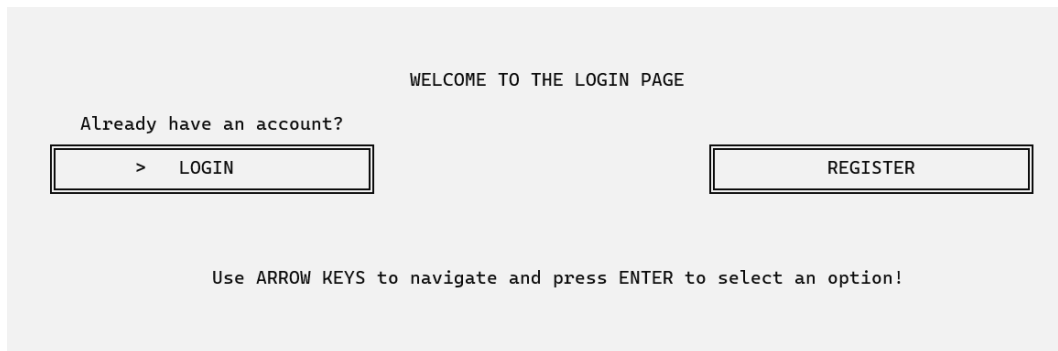
The same code is used for Down-sliding but the index i will now be used as the index of the column.

6 Our Other Features

6.1 Login and Register

In the login page, users can select one of two options: Login or Register. The user can navigate between these options using the arrow keys on their keyboard (Left and Right or 'a' and 'd'), and can select an option by pressing the Enter key.

The `int choice = 1;` line initializes the choice variable to 1, indicating that the Login option is initially selected. The code then enters a do-while loop that continues to execute until the program is terminated. Within the loop, the code first clears the screen and then prints the login page with the current choice option highlighted. Then, it waits for input from the user. If the user presses an arrow key, the program updates the choice variable to reflect the new selection. If the user presses Enter, the program executes the appropriate function based on the user's choice (`login()`, `registration()`)



The screenshot shows a terminal window with the following text:

```
WELCOME TO THE LOGIN PAGE

Already have an account?
> LOGIN                                REGISTER

Use ARROW KEYS to navigate and press ENTER to select an option!
```

Login page

If the user selects an invalid option, the program clears the screen and prints an error message asking the user to select from the available options. It then calls the `main()` function again to allow the user to make a new selection. The code also includes calls to two sound-playing functions which will play different sounds to indicate when the user has pressed an arrow key or pressed Enter.

The `login()` function allows the user to enter their username and password from their keyboard, which are then checked against a text file named "records.dat". If the username and password match, the user is logged in successfully and a message is displayed: "userID, your login is successful". If the username and password do not match, an error message is displayed: "Login error. Please check your username and password".


```
Please Enter Username and Password!
```

```
Username: Han
```

```
Password: ***
```

Login

The `registration()` function allows a new player to register by entering their username and password. Then, the username and password are saved in the "records.dat" file. `ofstream fs("records.dat", ios::app) :` This line of code is used to open the file named "records.dat" and data will be written to the end of the file. The `ios::app` flag ensures that any data written to the file is appended to the end of the file, rather than overwriting the existing contents of the file. After successful registration, a message is displayed: "Registration is successful".

```
Please Enter Username you want to use
```

```
Username: Nhi
```

```
Provide your Password for the next Login
```

```
Password: 1812
```

Register

6.2 Menu Page

In the menu page, there are four options: Start Game, How to play, Leaderboard and Quit, inside the Start Game mode, players can choose between Easy, Medium or Hard mode to play. Users can press arrow keys to navigate and press Enter to select an option

SetConsoleOutputCP(65001) is a function that sets the output code page of the console window to UTF-8, which is a variable-length encoding scheme for Unicode that can represent all characters in the Unicode standard.

The function starts by calling the printMenu function, which displays the menu options on the screen. It then sets the initial selected option to be the first option (selectedOption = 1) and moves the cursor to the corresponding position on the screen to display a '>' symbol next to the first option.

The function then enters a while loop, which runs continuously until the user selects an option. Within the while loop, the function listens for user input using the "inputKeyboard" function. Depending on the user input, the function updates the selectedOption variable and moves the cursor to the corresponding position on the screen to display the '>' symbol next to the new selected option.

If the user presses the enter key, the function plays a sound, and depending on the selected option, it calls different functions. If the user selects the first option, it calls the "playGame" function and then writes the updated list of players to a file using the "writeFile" function. It then calls the "selectingMenu" function recursively to display the menu again. If the user selects the second option, it calls the "howToPlay" function and then displays the menu again. If the user selects the third option, it calls the "leaderboard" function and then displays the menu again. If the user selects the fourth option, it calls the "exitGame" function and exits the program.

If the user inputs any other key, the function ignores it and continues listening for user input. This allows the user to cycle through two different options, such as selecting between different levels. The printMode(mode) function would display the current mode on the screen, as an arrow pointing to the currently selected option.



Welcome Screen



Menu

6.3 Print the game board information

```
+-----+
|                                     |
|               GAME INFORMATION    |
| Username: meo                     |
| Score: 15                         |
| Notification:                     |
|           YOU HAVE WON THE GAME   |
|   AUTOMATICALLY RETURN TO THE MENU |
| Press H for Move Suggestion       |
|                                     |
+-----+
```

Game information board

The interface displays the following information: "GAME INFORMATION" in blue text at the top of the box. The player's username in purple, the player's score in aqua , "Time left:" in red, "Notification:" in red and a message instructing the player to press the H key for move suggestions in green.

6.4 Time played

The Matching() function is responsible for counting the time it takes for the player to complete the game. First, it initializes a variable called start_time (represents the start time of the game) with the current time using the std::time function.

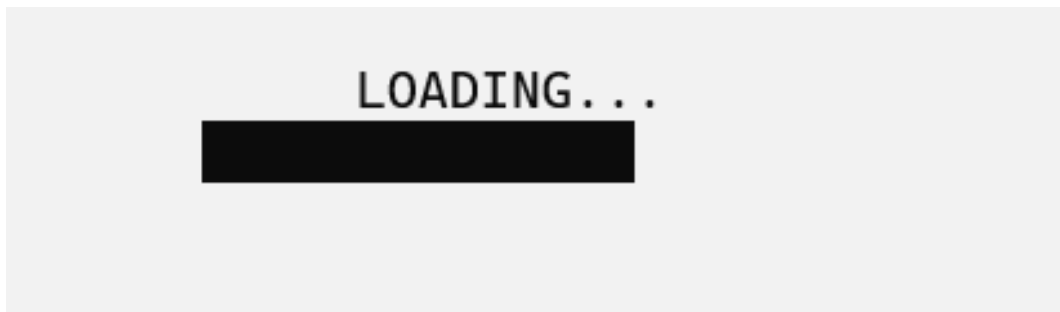
After the game is finished, the function initializes another variable called end_time with the current time. The std::difftime function is used to calculate the difference between the end_time and start_time in seconds, and then the result is stored in the 'time' variable of 'player' object.

6.5 Loading bar

The advantage of using a loading bar animation is that it can enhance the aesthetic appeal of the program, making it more visually pleasing and engaging for the user.

The function that creates a loading bar animation on the console screen using ASCII characters. It starts by clearing the console screen, then moves the cursor to the middle of the console screen and prints the text "LOADING...".

Next, the function moves the cursor to position (50, 10) and enters a loop that prints a series of 20 blocks (represented by the '■' character) with a short delay between each block using the Sleep function. This creates the appearance of a progress bar filling up over time. Finally, the function clears the console screen again, removing the loading bar animation from the screen.



Loading bar

6.6 Hide Cursor

This function is useful for hiding the console cursor during console applications or games, as it can help to improve the visual appearance of the program and prevent unwanted cursor flickering or distraction for the user. The function first obtains the handle to the console output using the "GetStdHandle" function. It then creates a structure called "CONSOLE_CURSOR_INFO" that contains information about the console cursor, including its size and visibility.

Next, the function sets the size of the cursor to 1 (the smallest possible size) using the "dwSize" member of the "CONSOLE_CURSOR_INFO" structure and sets the visibility of the cursor to false using the "bVisible" member of the "CONSOLE_CURSOR_INFO" structure). Finally, the function calls the "SetConsoleCursorInfo" function, passing in the console output handle and the "CONSOLE_CURSOR_INFO" structure to apply the changes to the console cursor.

7 Conclusion

Basically, our group has done relatively well in terms of completing the standard features of the project, fulfilling all of the technical requirements.

The process of analyzing and building the game has taught us valuable lessons and allowed us to gain more experience in team development and collaboration, enhancing our skills for future projects.

8 References

- [1] ASCII Art Archives, Buildings Places/Castles,
<https://www.asciart.eu/buildings-and-places/castles>.
- [2] ASCII Art Archives, Plants/Flowers,
<https://www.asciart.eu/plants/flowers>.
- [3] ascii.co.uk, Interactive ASCII Text Generator,
<https://ascii.co.uk/text>.
- [4] Simplilearn, Registration And Login Form In C++ | How To Create Login Page In C++,
<https://www.youtube.com/watch?v=yo3ImzEwP50>.
- [5] Admin, Lệnh tạm dừng và lệnh tạm dừng trong khoảng thời gian lập trình C/C++,
<https://tuicocach.com/lenh-tam-dung-va-lenh-tam-dung-trong-khoang-thoi-gian-vo>
- [6] Philips George John answered in May, 2012, Question: “getch and arrow codes”,
<https://stackoverflow.com/questions/10463201/getch-and-arrow-codes>.
- [7] Mohammad Usman Sajid answered in Jun, 2020, Question: “Move text cursor to particular screen coordinate?”,
<https://stackoverflow.com/questions/10401724/move-text-cursor-to-particular-screen-coordinate>
- [8] Microsoft Learn, COORD structure,
<https://learn.microsoft.com/en-us/windows/console/coord-str>.
- [9] Hot Examples, C++ (Cpp) GetStdHandle Examples,
<https://cpp.hotexamples.com/examples/-/-/GetStdHandle/cpp-getstdhandle-function.html>.
- [10] GeeksforGeeks, How to print Colored text in C++,
<https://www.geeksforgeeks.org/how-to-print-colored-text-in-c/>.
- [11] Yirkha answered in May, 2014, Question: “Most efficient possible code for printing a specific board C++”,
<https://stackoverflow.com/questions/23860284/most-efficient-possible-code-for-printing-a-specific-board>
- [12] Max O’Didily, How to Stop and Play Music Using C++,
<https://www.youtube.com/watch?v=z0ljIjBJvRI>.
- [13] Collaboration with other team: The development of the Loading bar function involved collaborating with team Minh Tue (22127440) and Hoai Nam (22127287) (Class 22CLC10).
- [14] Referring to Program Execution Instructions of Khai Minh (22127267) and Phuc Khang (22127180) (Class 22CLC08).