Course Project of SUSTech ME112
Introduction to Matlab
ME112 2024 Spring
February 19-June 16, 2024, SUSTech, China

Group 1

A TILE-MATCHING GAME IMPLEMENTED IN MATLAB

Fangmin Hou
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China, 518055
Email: 12111448@mail.sustech.edu.cn

Yuhan Liu
Department of Computer Science and Engineering
Southern University of Science and Technology
Shenzhen, China, 518055
Email: 12111811@mail.sustech.edu.cn

ABSTRACT

This paper presents the design and implementation of a tile-matching game, a popular puzzle game where the objective is to clear the game board by identifying and removing pairs of matching tiles. In this game, a valid move is recognized when the user selects two tiles on the same horizontal or vertical line, with the tiles between them being identical. This gameplay mechanic encourages strategic analysis of the game board to make successful matches. The game was implemented using MATLAB's built-in libraries, leveraging the platform's extensive GUI toolset to create an intuitive and user-friendly interface. The game board is represented as a 2D array data structure, and logical statements are utilized to determine the validity of each user input, ensuring that only legitimate tile matches are recognized and removed. To assist players, the developers have included a helper function that analyzes the current game state and suggests a valid move, if one is available. This feature can be particularly useful for players who may be struggling to identify the optimal tile combinations.

Keywords: Matlab, Game, Tile-Matching

1 INTRODUCTION

Tile-matching games have long captivated players of all ages, tracing their origins back to the early 1980s with the release of classic titles like Tetris and Columns.[9] Over the decades, these addictive puzzle games have evolved, incorporating new mechanics and visual aesthetics to challenge players in increasingly engaging ways.[3] The tile-matching game genre has become a staple of mobile and casual gaming, offering quick bursts of entertainment and mental stimulation.

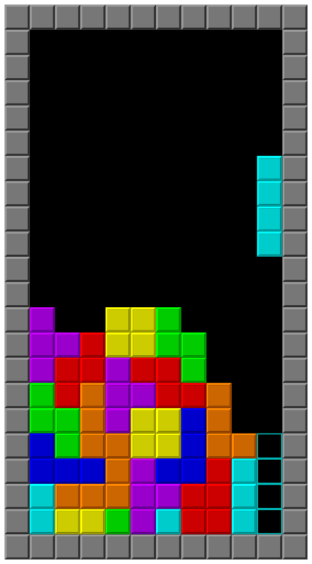


FIGURE 1. A typical Tetris game screen

This paper presents the design and implementation of a tile-matching game, a popular puzzle game where the objective is to clear the game board by identifying and removing pairs of matching tiles. In this game, a valid move is recognized when the user selects two tiles on the same horizontal or vertical line, with the tiles between them being identical. This gameplay mechanic encourages strategic analysis of the game board to make successful matches.

The game was implemented using MATLAB's built-in libraries, leveraging the platform's extensive GUI toolset to create an intuitive and user-friendly interface. The game board is represented as a 2D array data structure, and logical statements are utilized to determine the validity of each user input, ensuring that only legitimate tile matches are recognized and removed.

To assist players, the developers have included a helper function that analyzes the current game state and suggests a valid move, if one is available. This feature can be particularly useful for players who may be struggling to identify the optimal tile combinations.

2 METHODOLOGY

2.1 Graphical User Interface

The GUI is created using MATLAB's built-in graphics functions and components. The implementation provides a visually appealing and interactive interface for the tile-matching game. The code sets up the main figure, creates the game board axes, renders the tiles, and includes menu options and display messages to enhance the user experience.

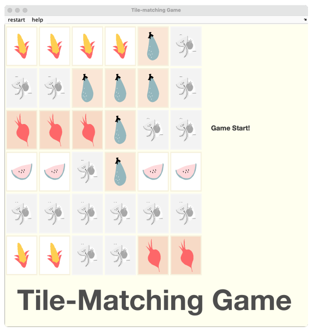


FIGURE 2. the GUI implementation

The game board is initialized with a size of l x w (6x6 by default) and number of tile kinds (6 by default) different tile types. The pairs parameter (20 by default) determines the number of tile pairs on the board. [8] The *init_game_board* function is called to create the initial game board state.

The main figure for the game is created with a size of 800x800 pixels, positioned at 750x250 on the screen. The figure has no title, no menu bar, and is non-resizable. The background color of the figure is set to a light yellow-ish color with RGB values of [1, 1, 0.9373].

Then an axes object is created within the main figure, taking up the entire figure area with a position of (0, 0, 1, 1). The x-limits are set to [40, 10100+105-40] and the y-limits are set to [40, 8100+85-40] to accommodate the tile size and margin. The axes are colored the same light yellow-ish color as the figure background, with the NextPlot property set to 'add' to allow for adding more objects. The layer is set to 'bottom' to ensure the tiles are on top, and the YDir property is set to 'reverse' to have the origin at the top-left corner. Finally, the X-Ticks and Y-Ticks are set to be invisible.

The GUI for the game includes two menu items. The first menu item is labeled "restart" and calls the function *init_game_board* to reset the game board. The second menu item is labeled "help" and is intended to call a solver function to provide assistance to the player.

The game board tiles are rendered using the image function. The tiles are placed on the axes with a margin of margin pixels between them. The drawPicHdl function is used to create the tile images, and the clickOnPic function is assigned as the ButtonDownFcn to handle tile clicks.

Three text objects is included to display various messages during the game. The first text object *count_text*, is meant to display the current game state. The second text object *invalid_text*, is intended to display messages about invalid moves. The third text object *game_status_text*, is used to display the game status, and is initially set to the message "Game Start!".

A title text object is created at the bottom of the game board with the text "Tile-Matching Game".

2.2 Game Logic

This *init_game_board* function is responsible for initializing the game board and generating the set of possible game pairs. The function first creates an empty game board of the specified length and width, represented as a 2D matrix of zeros. It then iterates through the specified number of game pairs, generating a random type of game element (kind) and a random starting position (x, y) on the board. For each pair, the function also randomly selects a direction (up, down, left, or right) and determines the ending position of the game pair based on the starting position and the selected direction. The function checks to ensure that the path between the starting and ending positions is clear (i.e., there are no other game elements in the way) before placing the game elements on the board. The function then stores the starting and ending coordinates of each generated game pair in the *generated_pairs* matrix, which can be used by other parts of the game logic. Finally, the function returns both the initialized game board and the generated game pairs, which can be used to set up the initial state of the game.

The *check_game_over* function is responsible for determining whether the game has ended and whether all game elements have been successfully eliminated from the game board. The function iterates through all the cells on the game board, checking the value of each cell. If a cell contains a non-zero value (i.e., a game element), the function sets *point_exist* to true. It then checks the cells adjacent to the current cell to see if there are any other game elements of the same type that could be paired. If no such adjacent cells are found, the function sets *all_eliminated* to false, indicating that there are still some game elements that have not been eliminated. Finally, the function sets the *game_over* variable to true4 if either *all_eliminated* is true (indicating that all game elements have been eliminated) or *point_exist* is false (indicating that there are isolated game elements that cannot be paired). The function then returns the *game_over* and *point_exist* values, which can be used by other parts of the game logic to determine the current state of the game.

A *solve_game* function is implemented, which provides the core logic for automatically finding a solution to the game based on the given inputs. The function first initializes the *solution_nums* variable to 0 and creates a *solution_pairs* matrix with the same number of rows as *generated_pairs*, but with 4 columns to store the solution coordinates. It then iterates through each pair in the *generated_pairs* matrix, checking if the coordinates are valid (non-zero). If the pair is valid, the function determines if the x-coordinates are the same (i.e., it's a vertical line) or if the x-coordinates are not the same (i.e., it's a horizontal line), and then finds the start and end points of the solution along the appropriate axis. If a solution is found (i.e., the start and end points are different), the function increments the *solution_nums* variable and stores the solution coordinates in the *solution_pairs* matrix. Finally, the function returns the *solution_pairs* matrix and the *solution_nums* value, which can be used by other parts of the larger application to solve the game or puzzle.

3 DATA ANALYSIS

$$GAME\_BOARD = \begin{bmatrix} 3 & 3 & 3 & 1 & 0 & 1 \\ 6 & 6 & 1 & 1 & 0 & 1 \\ 2 & 1 & 1 & 4 & 4 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 4 & 4 & 4 & 4 \\ 4 & 4 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We represent the game board as the matrix provided above. Each element within the matrix encodes the type of game piece occupying the corresponding cell. Specifically, a value of 0 indicates that the cell is empty, devoid of any game tile. Conversely, non-zero integer values denote the particular kind or type of game piece situated in that cell. The figure 3 shows the corresponding game board of the matrix. The presented matrix representation has the x and y axes interchanged compared to the

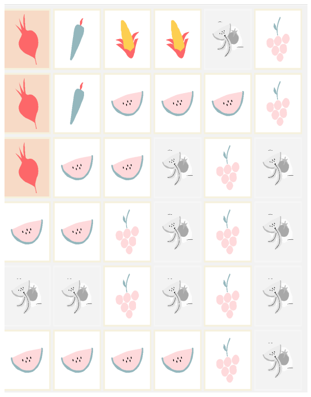


FIGURE 3. the corresponding game board

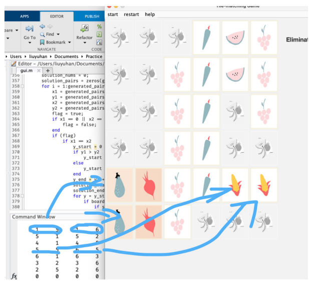
typical convention. That is, the row index corresponds to the y-coordinate, while the column index represents the x-coordinate of the game board.

In the analysis of the *check_board_play* function, the input provided by the users is carefully examined. The data analysis process can be described as follows. Firstly, the algorithm checks whether the x or y coordinates of the two input points are identical. If this condition is met, the next step is to verify whether the two points are of the same kind by inspecting the value stored in the corresponding cell of the matrix. Thirdly, the function proceeds to check whether all the intermediate points between the two input coordinates are also of the identical kind. Finally, the analysis outputs the determination of whether the play is valid and the number of points that would be eliminated as a result of the move.

The *init_game_board* function sets up the initial game board in a randomized manner. First, it creates a 2D matrix of all zeros, with dimensions based on the given length and width. Next, it generates a set of random positions on the board, equal to the number of required tile pairs. For each of these positions, it randomly determines the direction (horizontal or vertical) and the number of tiles in the pair. The function then places the tile pairs on the board, checking that there are no other tiles already occupying the target positions and that the tile pair does not extend beyond the boundaries of the game board. If both conditions are met, the tile pair is placed on the board. Otherwise, the function skips that position and moves on to the next one. Once all the tile pairs have been placed, the function returns the populated game board matrix.

The *check_game_over* function determines whether the game has reached a state where there are no more valid tile pairs left on the board. It does this by iterating through each cell on the game board. For each cell, it checks if there is another tile of the same type directly below it (in the same column) or to the right of it (in the same row). If such a matching tile is found, it means there is still at least one valid tile pair on the board, and the game is not over. If the function completes its loop through all cells without finding any matching tiles, it means there are no more valid tile pairs left, and the game is over. In this case, the function returns True to indicate the game is complete. Otherwise, it returns False to indicate the game is still in progress. This approach allows the function to efficiently check the entire game board state and determine whether any valid tile pairs remain, without the need for any additional data structures or complicated logic.

The *solve_game* function takes the current game board and the set of generated tile pairs as input. Its goal is to determine the start and end positions of any remaining valid tile pairs on the board. To accomplish this, the function iterates through each of the generated tile pairs. For each pair, it checks if the tiles in that pair are still contiguous on the game board - that is, if the tiles are still adjacent to each other in the same orientation (horizontal or vertical) as when they were initially placed. If the function finds that a tile pair is still contiguous, it records the start and end positions of that pair. These positions are then output as the solution. If the function completes its check of all tile pairs without finding any contiguous pairs, it means there are no more valid tile pairs remaining on the board. In this case, the function returns an empty set to indicate that the game has been solved. This approach allows the *solve_game* function to efficiently identify any remaining valid tile pairs on the board and report their locations, providing the information needed to complete the game.



```
        solution_nums = 0;
        solution_pairs = zeros(g
        for i = 1:generated_pair
            x1 = generated_pairs
            y1 = generated_pairs
            x2 = generated_pairs
            y2 = generated_pairs
            flag = true;
            if x1 == 0 || x2 ==
                flag = false;
            end
            if (flag)
                if x1 == x2
                    y_start = 0
                    if y1 > y2
                        y_start
                    else
                        y_start
                    end
                    y_end = y
                    solution_
                    solution_end
                    for y = y_st
                        if board
                            if s
```

```
     1     5     1     6
     5     1     5     2
     4     1     4     5
     5     5     6     5
     6     1     6     3
     3     2     3     6
     2     5     2     6
     0     0     0     0
```

FIGURE 4. the output solutions

As shown in the figure 4, the output is a matrix. Each row of the matrix indicates the x and y values of the start positon and the x and y values of the end postion. The figure shows how they match the tiles in the game board.

4 CHALLENGES AND SOLUTIONS

One of the challenges we faced was efficiently updating the game board after each tile-matching event. Our initial implementation involved iterating through the entire board and comparing the current state to the previous state, in order to identify and update the tiles that had changed. However, this approach proved to be inefficient, as it required a large number of nested loops, which had a negative impact on the overall performance of the game. To address this issue, we optimized the update process by only updating the tiles that had been modified, instead of scanning the entire board. This approach significantly improved the performance of the game, as it reduced the computational overhead associated with the board update process.

Another challenge we encountered was the design of the game's graphical user interface (GUI). Initially, the GUI lacked clarity and failed to provide adequate information about the game's status to the player. To resolve this problem, we added several text objects to the interface, which helped to clearly communicate the current state of the game to the player. Moving forward, we recognize the importance of a well-designed GUI, especially during the initial stages of the development process. In future projects, we will place a greater emphasis on the design of the user interface, ensuring that it is intuitive, informative, and visually appealing to the player.

5 INSIGHTS

We have developed similar games in the Java course. Now we are using Matlab to do the same thing again. We find a lot of differences here.

5.1 Application Domains

MATLAB is a programming language that was created for scientific computing, as evident from the "matrix" in its name. Many of MATLAB's features are designed for numerical computation, such as its 1-based indexing and extensive matrix operations. MATLAB's design is particularly well-suited for use by scientists and researchers. It is easy to pick up, with mathematical expressions that read quite naturally. It provides a wealth of tools for numerical computation, like the ode45 function we learned about in class. MATLAB also excels at data visualization, and its toolbox ecosystem is remarkably comprehensive.

In contrast, Java is a general-purpose, object-oriented programming language, unlike MATLAB which is more specialized. Java has a much broader range of applications - you can find Java everywhere, from the JetBrains IDEs we're familiar with, to the Android operating system and the countless apps built on it, to the server-side SpringBoot framework that powers countless internet applications, to the ubiquitous presence of Java in the world of microservices.

Due to their different design goals and target application domains, the experiences of programming in MATLAB and Java can vary significantly.

5.2 Graphical Libraries

Matlab's graphics library is designed for the convenience of scientific computing, so it is relatively simple and easy to learn, with a low learning curve. Compared to Java, Matlab provides more data visualization functions such as surf and plot, which are very suitable for scientific computing. However, due to its focus on scientific computing, Matlab lacks the flexibility of Java.

Java, on the other hand, has many graphics libraries such as JavaFX. Since Java is ubiquitous, including in Android applications, Java's graphics libraries are very rich and versatile. However, the learning curve is also greater due to the wide range of applications it supports, compared to Matlab's focus on scientific computing.

In summary, Matlab is designed for scientific computing and lacks general-purpose capabilities, while Java is a general-purpose programming language that does not support scientific computing as well as Matlab.

5.3 Language Types and Speed

Matlab is an interpreted language, and interpreted languages are generally easier to learn compared to compiled languages. Many scientific practitioners may not have extensive programming experience, so Matlab, as a tool serving them, being an interpreted language is only natural.

Unlike compiled languages, interpreted languages only report errors when they are encountered during runtime, as the interpreter is responsible for interpreting the code. This means that interpreted languages tend to be slower in execution speed.

On the other hand, Java is a general-purpose programming language that prioritizes speed. [4] Java is a compiled language, where the code is first compiled into machine code, and then executed by the Java Virtual Machine (JVM). This compiled approach makes Java significantly faster than interpreted languages, and errors can be detected at the compile-time rather than runtime.

6 TEAM COLLABORATION

Here is the team collaboration.

TABLE 1. TEAM COLLABORATION

Name	Responsibilities
Fangmin Hou	Display; UI/UX Design and Implementation
Yuhan Liu	Game Logic and Solver Tool Implementation

This division of responsibilities was based on our individual strengths. Fangmin Hou has a good aesthetic sense, so she wanted to handle the UI portion and is particularly interested in the front-end work. Yuhan Liu is more interested in systems, so he wanted to explore the backend logic behind a tile match game system, and thus chose to focus on the backend.

References

[1] Dorothy C Attaway. *Matlab: a practical introduction to programming and problem solving*. Butterworth-Heinemann, 2013.

[2] Mark Austin and David Chancogne. *Introduction to engineering programming: in C, Matlab and Java*. John Wiley & Sons, 1999.

[3] *candy crush saga wiki*. URL: https://candycrush.fandom.com/wiki/Candy_Crush_Saga.

[4] *comparison of matlab and other oo languages - matlab simulink*. URL: https://www.mathworks.com/help/matlab/matlab_oop/matlab-vs-other-oo-languages.html.

[5] Ceren Cubukcu, Zeynep Behrin Guven Aydin, and Ruya Samli. "Comparison of C, Java, Python and Matlab programming languages for Fibonacci and Towers of Hanoi algorithm applications". In: *Boletim da Sociedade Paranaense de Matemática* 41 (2023), pp. 1–7.

[6] Desmond J Higham and Nicholas J Higham. *MATLAB guide*. SIAM, 2016.

[7] *MATLAB documentation*. URL: https://www.mathworks.com/help/matlab/index.html?s_tid=hc_panel.

[8] *tile matching picture material*. Dec. 2020. URL: https://blog.csdn.net/qq_40930634/article/details/111923895.

[9] *tile-matching wiki*. June 2024. URL: https://en.wikipedia.org/wiki/Tile-matching_video_game#Significance.

[10] Daniel T Valentine and Brian H Hahn. *Essential MATLAB for engineers and scientists*. Academic Press, 2022.