



JUSTUS-LIEBIG- UNIVERSITÄT GIESSEN

Justus Liebig University Giessen
Faculty 06 Department of Sports Science
Human Movement Analytics
WS 2023/2024

Authors:
Yeganeh Mohammadi
Sabrina Miller

Emails:
yeganeh.mohammadi@sport.uni-giessen.de
sabrina.miller@sport.uni-giessen.de

Matriculation Number:



Module:
MA-BMB-05 Seminar Computer Programming
in Human Movements Analytics

Lecturer:
M. Sc Lea Junge-Bornholt

Submission date:
10.03.2024

Contents

Introduction.....	3
Game overview (Snake & Ladder)	3
Fundamental ideas	3
Main menu	4
Implementation	4
User manual	4
Challenges.....	4
User interface.....	4
Implementation	5
User manual	9
Challenges.....	9
Mini-games	9
Order memorizer.....	9
Fundamental ideas	10
Implementation	10
User manual	11
Challenges.....	11
Memory game	12
Fundamental ideas	12
Implementation	12
User manual	14
Challenges.....	15
Sudoku	15
Fundamental ideas	15
Implementation	16
User manual	17
Challenges.....	17
Future Development.....	17
Conclusion	17

Introduction

The Snake and Ladder game has been a popular board game for generations, and this project aims to bring the classic game to the digital realm, providing an interactive and entertaining experience for players. The main goal is to create an attractive and interactive game that preserves the essence of classic board games while using computer programming capabilities.

Snake and Ladder has an attractive and user-friendly visual interface that allows players to roll dice and move their game pieces. In addition to the main game, for more challenge, three mini-games will be played, which will provide additional entertainment for the players.

This report will outline the technical details of the game's implementation, including the programming language and framework used, as well as the design and architecture of the game. Additionally, it will discuss the three mini-games incorporated within the project, enhancing the overall gameplay experience.

Game overview (Snake & Ladder)

The Snake and Ladder game is a classic board game played on a square grid featuring numbered squares. The objective of the game is to navigate through the board from the starting point to the finish line, represented by the highest-numbered square, using a dice roll to determine the number of steps the player can take. The game incorporates elements of chance and strategy, as players encounter snakes that send them backward, and ladders that allow them to progress forward.

To play the game, players take turns rolling a dice and move their game piece accordingly. If a player lands on a square with the base of a ladder, they climb the ladder, advancing to a higher-numbered square. Conversely, if they land on a square with the head of a snake, they slide down the snake, moving to a lower-numbered square. The first player to reach or exceed the finish line square wins the game.

Fundamental ideas

We initiated the project by conceptualizing a menu that enables players to input their names, select the color of their game pieces, and access the high score lists.

In order to create an engaging and visually appealing game interface for the main game, our focus was on designing a visually distinct board game with snake and ladder figures. The design approach sought simplicity while ensuring a productive and enjoyable gameplay experience. As part of this design, we considered the implementation of a dice rolling button that triggers an animation simulating the rolling of a dice when pressed.

Furthermore, we explored the idea of displaying player information through fields in the main game. After players enter their names and choose their colors in the menu, these details will be showcased on these fields. Additionally, their current positions on the game board will be dynamically updated, providing real-time feedback on their progress.

To enhance the game's level of challenge, we have developed the concept of a "Risk Button." This feature allows players to press the button during gameplay, triggering a random mini-game. The difficulty level of the mini-game can be chosen by the player. Upon completion of the mini-game, players will be rewarded based on the level of difficulty they selected. This addition adds excitement, variety, and additional rewards to the gameplay experience.

Main menu

In designing the menu for our game, we prioritized creating a visually appealing and user-friendly interface that enhances player engagement from the moment they launch the game. The menu features are designed with vibrant colors and navigation elements, ensuring that players can easily access various options and settings.

One key feature of our menu is the ability for players to personalize their gaming experience by entering their name. This personalization not only creates a sense of ownership but also allows players to feel more connected to the game. Another aspect of our menu is the inclusion of a color selection option for game figures. We understand that players have diverse preferences, and offering the ability to choose the color of game figures adds an extra layer of customization.

In the future, we plan to enhance the user experience by expanding the options in our main menu. This includes the incorporation of language selection functionality, making the game more accessible to a broader audience.

Implementation

The provided callback function, named "LetsPlayButtonPushed()", is designed to execute when the "Let's Play" button is activated within the application. Upon activation, the function proceeds to save the pertinent information of both players into separate (.mat) files. In addition, the function checks if both players have entered their name and then start the main game.

The provided callback function, named "ColorDropDown1ValueChanged()", is triggered whenever there is a change in the selected color by Player 1 in the dropdown menu. Upon activation, the function updates the variable storing the selected color for Player 1. Additionally, the function dynamically adjusts the available color options for Player 2 to prevent both players from selecting the same color. It accomplishes this by removing the selected color from the list of available options and updating the dropdown menu accordingly.

The "ColorDropDown2ValueChanged()" callback function performs the same task as the "ColorDropDown1ValueChanged()" callback function. It dynamically updates the available color options for the other player to prevent duplicate color selections and ensures consistency in the color choices between both players.

User manual

- Enter your name in the designed field.
- Choose the color for your game figure from the available options.
- Press the "Let's Play" button to start the main game.

Challenges

Our main challenge while coding the menu was making sure both players could not pick the same color. We had to carefully adjust the available color options to avoid this. It took some trial and error to get it right, but we eventually found a solution that worked well and made the game fair for everyone.

User interface

The user interface of the main game incorporates several features to enhance the gaming experience. These features include a game board, menu, and info buttons, as well as rolling dice animations. Additionally, images are utilized to represent snake and ladders elements. Labels and

fields are employed to display players' names and positions, while icons are used to indicate risky actions. The color schemes chosen for the main game are relevant to the snake and ladders theme, adding visual cohesion.

For the mini-games, various elements are utilized to create excitement and a sense of risk. Each mini-game app employs a distinct color scheme to differentiate them. The mini-games' user interfaces consist of buttons, edit fields, uiaxes, labels, and a variety of images. To ensure accessibility, the option to resize the app is enabled, accommodating different window styles and PCs. This decision was made to create a user-friendly experience for all users.

Implementation

The "generateBoard()" function plays a vital role in the implementation of the mini-game by creating a properly formatted game board with labeled fields. It dynamically generates the game board by utilizing the generateBoard() function, which creates a matrix of uiLabels representing each individual field on the board. The positioning of the labels is precisely determined by nested for-loops, iterating over each row and column of the game board. The x and y positions of each label are calculated based on the row and column indices, the button size, and the spacing between labels.

To enhance the visibility and aesthetics of the game board, the background colors of the labels were modified to create an appealing board pattern. This was achieved through the implementation of an alternating background color scheme. The sum of the row and column indices is checked, and if the sum is even, a light orange color with transparency 0.5 is used. Conversely, if the sum is odd, a peach color with transparency 0.5 is used.

The labels representing the board fields are customized using various name-value arguments. Specifically, the text of each label is set to the corresponding order number from the "order_numbers" matrix. The "order_numbers" matrix is generated in a separate function called "numberOrderMatrix()". Initially, the "order_numbers" matrix is set up with empty values using NaN. The size of the matrix is determined by the number of rows and columns. The starting point for populating the matrix is chosen as the bottom-left corner, beginning with the last row and the first column.

Next, the matrix is filled with numbers from 1 to 100 in a snake-like pattern. This is achieved through a loop that iterates 100 times. In each iteration, the current number is assigned to the element at the current row and column in the "order_numbers" matrix. To create the snake-like pattern, the function checks if the current row is even or odd. If the row is even, the column index is increased to move to the right, until the last column is reached. In that case, the function moves up to the previous row. If the row is odd, the column index is decreased to move to the left, until the first column is reached. Again, in that case, the function moves up to the previous row.

The "loadDataFromUserInput()" function is responsible for loading the data entered by the user in the main menu and displaying it in the game interface.

The "figureStartPosition()" function is responsible for positioning the figures on the start field (field1) in the game. It performs several important tasks to set up the game's initial state. Firstly, it identifies the position of the start field by calling the "findPositionOfFigure()" function, ensuring that the figures are placed correctly on the start field. Next, the function creates the player's figures using the uiimage function. Each figure is associated with an image source that corresponds to the

player's chosen color. Additionally, the figures are given a specific position on the start field and assigned an "ImageClickedFcn" callback function, which determines the action to be taken when the figure is clicked.

After positioning the figures, the function proceeds to check if there are two figures in the same field by calling the "check2FiguresInOneField()" function and passing in the start position. This step ensures that the figure images are both visible on the game board and that they do not overlap.

The "check2FiguresInOneField()" function is responsible for verifying whether two figures occupy the same field in the game. If two figures are found in the same field, the function adjusts their size and position to ensure they can fit beside each other without overlapping. The function begins by comparing the positions of the first player's figure and the second player's figure to determine if they share the same position. If the figures are indeed in the same field, they need to be adjusted. On the other hand, if the figures are not in the same field, their positions are reset to their original positions using the "findPositionOfFigure()" function. This ensures that the figures are correctly positioned within their respective fields.

The "findPositionOfFigure()" function is responsible for determining the position of a specific field on the game board using the userdata. This information is important for correctly placing figures on the corresponding fields. The function starts by using the findobj() function to search for the field within the UIFigure based on its tag. The field is identified by its type (ui-label) and the UserData property, which is set to the string representation of the tag_of_field. Once the field is located, its position is retrieved using the position property and stored in the field_position variable. The field_position contains information about the field's position and, size. Next, the x and y positions of the field are extracted from field_position and stored in x_position and y_position, respectively. Finally, the figure_position is formed using the x_position and y_position values and returned as the output of the function. This figure_position represents the desired position for placing a figure on the field.

The "moveFigure()" function is a crucial and challenging part of the game implementation as it handles the movement of a player's figure on the game board based on the dice roll and manages event fields, such as snakes and ladders. The function begins by checking whose turn it is, either Player 1 or Player 2. If it's Player 1's turn, the function verifies if there is no figure of Player 2 on the same field. If there isn't, it makes the field label visible again. Next, the function calculates the new position of the figure by adding the current position to the number rolled on the dice. It updates the figure's position on the game board and removes the text from the new position field. The function then checks if the new field is a special field, such as a snake or ladder.

If it is, the function checks if there is no figure of the other player on the same field. If there isn't, it makes the field label visible again. It also updates the figure's position based on the post-event-fields value. Finally, the function adjusts the figure's position on the game board and removes the text from the new position field, completing the movement of the player's figure on the game board based on the dice roll and handling any necessary event fields.

The "whoBegins()" function is responsible for randomly determining which player gets to begin the game. The randi([1,2]) function is then used to generate a random integer between 1 and 2, representing the player who will begin the game.

The "numberOfFieldOnOff()" function, within the app object, allows for the control of text visibility on a field. By passing different values for the text parameter, the function can either show or hide text on the specified field. Additionally, it employs the UserData property of the field to ensure that the correct field is located, even if the text has been removed.

The "updatePositionField()" function updates the position display for each player on the game board. It retrieves the field associated with the player's position and updates the position display button with the corresponding background color and position number.

The "updateWhoseTurnIsItLabel()" function updates the label that informs the players whose turn it is in the game. It uses the players' names to personalize the message displayed in the label.

The "figure2Clicked()" function is similar to the "figure1Clicked()" function, but it specifically handles the actions and logic associated with Player 2's moves or interactions with the game board.

The "figure1Clicked()" and "figure2Clicked()" functions handle the behavior when the user interacts with a specific figure during Player 1's turn in the mini-game. It disables the RiskButton, moves the figure, checks for multiple figures in one field, updates the position, switches the turn to Player 2, enables the RollDiceButton, hides the dice image, updates the turn label, and resets relevant variables. This function ensures smooth gameplay and proper turn management.

The "checkWinning()" function determines if a player has won the game by reaching field 100. It displays a congratulatory message and disables relevant buttons and figures accordingly. Additionally, it ensures that the figures do not move beyond the winning field, giving the player another chance to roll the dice in subsequent turns.

The "adjustDiceRange()" function adjusts the range of numbers on the dice based on the difficulty level of the mini-game. It loads relevant data, checks the difficulty level, and assigns an appropriate dice range to allowed_dice_range. This adjustment influences the possible outcomes of the dice roll during gameplay.

The "rollDice()" function is a crucial component of the mini-game, responsible for simulating the process of rolling dice. This function is designed to handle both the regular dice and a special dice. When the player clicks the "Roll the Dice" button, the function determines whether it's the first or second roll.

During the first roll, the function initiates a rolling dice animation by displaying a GIF on the screen. This animation creates a visual effect of dice being rolled. After a brief pause, the animation disappears, indicating the end of the roll. At this point, a random number between 1 and 6 is generated, representing the result of the regular dice roll. The corresponding dice image is then displayed on the screen, providing a visual representation of the rolled number. Additionally, the Risk Button is enabled, allowing the player to proceed with their next action.

In the case of the second roll, the function follows a similar process. The current dice image is hidden, and the rolling dice animation is displayed again. After a brief pause, the animation disappears. This time, a random number between 1 and a dynamically adjusted dice range is generated. This adjusted range is determined by a function called "adjustDiceRange()", which modifies the range based on the difficulty level of the mini-game. The special dice number is then obtained, and its corresponding image is displayed on the screen. The rolled numbers from both

the regular and special dice are added together to calculate the total dice number for the mini-game.

The "startupFcn()" function is a callback function that is executed when the application starts up. It performs several important tasks to initialize the game and set up the necessary elements. First, the function adds a path to the "main_game_pics" directory. This directory likely contains image assets used in the game.

Next, the function calls "loadDataFromUserInput()", which is a custom function responsible for loading relevant data from user input. The "numberOrderMatrix()" function is then called. This function likely generates a matrix that determines the order of numbers on the game board or assigns unique numbers to each cell on the board. Following that, the "generateBoard()" function is invoked. This function is responsible for creating the game board, which could involve creating cells, setting up the layout, and positioning the necessary game elements.

The subsequent lines of code use the uistack() function to adjust the stacking order of various image elements on the game board. This ensures that certain images, such as snake and ladder graphics, are displayed on top of other elements. The "figureStartPosition()" function is then called. This function likely determines the starting position of the game figures or tokens on the board, ensuring that they are correctly positioned at the beginning of the game. Finally, the "whoBegins()" function is invoked. This function determines which player begins the game, possibly through a random selection or based on specific game rules.

The "RollDiceButtonPushed()" function is a callback function that is executed when the "Roll Dice" button is pushed by the player. First, the function disables the "Roll Dice" button by setting its Enable property to 'off'. This prevents the player from clicking the button multiple times before the dice roll is completed. By disabling the button, the function ensures that only one dice roll can be performed at a time, maintaining the integrity of the game.

Next, the function calls the "rollDice()" function. This function is likely defined elsewhere in the code and is responsible for simulating the rolling of the dice in the game. By invoking this function, the player initiates the dice roll process, which generates random numbers and determines the outcome of the roll. The "rollDice()" function may involve displaying rolling dice animations, updating the game state, and performing any necessary calculations based on the rolled numbers.

The "RiskButtonPushed()" function is a callback that handles the player's interaction with the "Risk Button". When pressed, the function prompts the player to choose a difficulty level for the mini-game and saves it. It then selects a random mini-game, waits for it to finish, and checks the result. If the player wins, the "Roll Dice" button is enabled again; otherwise, their figure is not allowed to move. This function adds excitement and variability to the game by incorporating mini-games and their outcomes into the main gameplay.

The "ImageInfoClicked()" function is a callback that is triggered when the player clicks on an image, presumably to obtain information or instructions about the Snake and Ladder game. In addition, there is another function mentioned in your code snippet called "ImageMenuClicked()". This function is a callback that is executed when the player clicks on a menu image.

User manual

- The main game screen will appear, ready for you to start the game.
- To access the game instructions, tap the 'Info' icon located within the app. The game instructions will be displayed, providing you with a clear understanding of the gameplay and rules.
- Press the 'Roll the Dice' button on the game interface to initiate a dice roll.
- The dice will roll and determine the number of spaces around which your game piece moves.
- Each player can easily track their positions on the game board. Above the game board, there is a designated button for each player that displays their current position.
- For those seeking more challenge, the game offers a "Risk It" button.
- Pressing the "Risk It" button triggers a pop-up question box, asking you to choose the difficulty level.
- Based on the difficulty level you select, you will be presented with random mini-games such as Order Memorizer, Memory Game, or Sudoku.
- If you successfully complete the mini-game, you will be rewarded accordingly.
- In the event that you lose the mini-game, you are 'punished' with not moving your figure at all.
- you need to click on your figure to complete each move, even if you lose the mini-game and aren't allowed to move at all.
- If you ever need to return to the main menu, simply press the menu icon.

Challenges

One significant challenge we encountered was determining the position of the figure on the game board. This task proved to be quite complicated, particularly when dealing with multiple figures or a complex game board layout. Another difficulty we faced was managing the parameters when two figures occupied the same field. It was crucial to update the parameters appropriately to reflect the presence of two figures and ensure they returned to their normal size when one of the figures moved.

A huge challenge emerged when attempting to utilize tags for labels. We discovered that tags are not available for uilabels, which was an obstacle to achieving the intended functionality. However, a creative solution was devised to overcome this limitation. The user data property of the labels was leveraged to store the order numbers as strings. This workaround effectively provided the desired functionality, allowing for the proper organization and identification of labels despite the absence of tags.

Another challenge was implementing a wait function that ensured the mini-game was fully closed before reloading the data in the main game. This was necessary to maintain data integrity and prevent any conflicts or inconsistencies between the two parts of the game. Waiting for the mini-game to close ensured that the main game could accurately reload the necessary information.

Mini-games

Order memorizer

In this game, players will encounter a sequence of quickly changing vibrant colors. Their task is to reproduce this sequence accurately, relying on memory and attention to detail.

What adds to the excitement is that the "Order Memorizer" mini-game can only be played once. It's triggered within the main game, Snake and Ladder, when players decide to take a risk. If they succeed in the "Order Memorizer" mini-game, they'll earn rewards based on the chosen difficulty level.

Fundamental ideas

We have developed an exciting mini-game called "Order Memorizer". To create the game, we decided to implement a grid with 9 fields where the gameplay takes place. Each field will randomly light up, and the number of lit fields will vary based on the difficulty level selected by the player. To keep track of the order in which the fields light up, we opted to use linear indexing. Additionally, we have considered including an info icon to provide instructions to the players, as well as implementing a function to automatically close the mini-game at the end.

Implementation

We require four main functions for this mini-game: "generateOrder()", "switchColourGreen()", "checkIfUserPushedTheRightButton()", and "saveResultOfGame()".

The first function we implemented was the "generateOrder()" function, which generates a random order of button presses using the randperm function. It determines the number of buttons that players need to remember based on the chosen level. For instance, if the level is set to easy, players are required to remember four buttons. If the level is medium, they need to remember six buttons. And if the level is hard, they have to remember eight buttons.

The next function is "switchColourGreen()", which changes the background color of the specified button to green, providing visual feedback to the user when a button is pressed correctly. After a 1-second pause, the button's color is reverted back to white.

Following that, the "checkIfUserPushedTheRightButton()" function is implemented to verify the player's answer. In this function, first, it is checked if any buttons are still green to address the issue of players clicking the next button too quickly. Then, the pressed button is compared with the expected button based on the user's progress. If the correct button is pressed, the user's progress is updated, and the pressed button is highlighted in green. If the user has completed all steps, a message box is displayed to notify the user that they have won. The "result_memorizer" variable is set to 1 to indicate the game result, and the "saveResultOfGame()" function is called to save the game information. Another message box instructs the user to close the app to continue with the main game. Conversely, if an incorrect button is pressed, the pressed button is highlighted in red, all buttons are disabled, and a message is displayed to indicate a loss.

Our final function is "saveResultOfGame()". This function is responsible for saving the important variables that will be used in the main game after the completion of the mini-game. It stores the difficulty level chosen by the player and the result of the game (whether the player won or lost).

For this mini-game, the use of callbacks is essential. The "ButtonStartPushed()" callback function is triggered when the start button is pressed. It calls the "generateOrder()" function, passing in the difficulty level chosen by the player. Next, the "switchColourGreen()" function is invoked for each button in the generated order, changing their background color to green. A pause is introduced

after displaying the sequence, and a message box appears to inform the user that they can start pressing the buttons. Finally, the enablement of all buttons is set to "on" to enable user interaction.

Additionally, we have `ButtonPressedCallbacks` (one for each button). These callback functions are triggered when the user presses a button. Each callback function calls the `"checkIfUserPushedTheRightButton()"` function, passing in the corresponding button number to determine if it was pressed correctly.

Lastly, we have the callback for `"ImageInfoClicked,"` which is used to provide game instructions to players. When this callback is triggered, it displays a message box containing the instructions for the game. The `"UIFigureCloseRequest()"` callback function is utilized to handle the event triggered when the user attempts to close the user interface (UI) figure associated with the mini-game. This callback function ensures that the main game is notified when the mini-game is closed, enabling the main game to resume its execution.

User manual

To play "Order memorizer" follow these instructions:

- Press the "Start" button to initiate the game.
- Watch as the cards on the screen change color randomly, based on the chosen difficulty level.
- Pay close attention to the sequence in which the cards change color, as you'll need to replicate this sequence later.
- After a brief period, the cards will stop changing color, indicating it's your turn to select the cards in the correct order.
- Click on the cards in the order you believe matches the initial sequence of colors.
- If your selection matches the correct order, you'll receive a message indicating your victory. Additionally, based on the chosen difficulty level, you will be rewarded.
- If your selection is incorrect, you will be notified of your loss and won't receive any reward.
- For additional game instructions, click on the "Info" button.

Challenges

A major challenge we faced was deciding whether to use `randi` or `randperm` for generating a random order of button presses. The decision hinged on the requirement of allowing repetition. While `randi` enables the generation of random numbers with possible repetitions, `randperm` ensures a unique order.

We also encountered the challenge of creating a general function that could handle the callbacks for all the buttons in the game. This was important to avoid duplicating code and improve maintainability. By designing a modular function that accepts the button number as a parameter, we were able to perform the necessary actions, such as checking correctness, updating progress, and providing visual feedback, for any button in the game.

Closing the app automatically posed another challenge for us. Initially, our plan was to create a function that would automatically close the app when the game was finished. However, we encountered complications in implementing this idea. As a result, we decided to change our approach and instead allow the player to manually close the app when they are done playing.

Another important challenge we faced was implementing a mechanism to check if a button was still lit up. This was challenging because we wanted to prevent players from clicking the next button too quickly. To address this, we developed a system that tracked the state of each button to determine if any previously clicked buttons were still lit up.

Memory game

In this mini-game, players are presented with a sequence of pairs of similar images. The challenge is to find and match these pairs accurately. Players must pay close attention to the images on the cards and try to remember their positions.

What adds to the excitement is that this mini-game can only be played once. It is triggered within the main game, Snake and Ladder, when players face a decision. If they choose to take a risk, they may encounter the "Memory game." Successful completion of this mini-game results in rewards corresponding to the chosen difficulty level.

Fundamental ideas

The "Memory Game" utilizes buttons with images underneath that are revealed when the player clicks on the button above the image. Implementing this concept requires several functions. The setup function is responsible for arranging the cards on the game board, assigning images to each card, and ensuring they are initially hidden from view.

To enhance the gameplay experience, we have identified the need for specific functions to handle various aspects of the "Memory Game". When a player clicks on a card, it is important to have a function that effectively shows the image associated with that particular card. Additionally, a separate function is required to compare the images on the chosen cards. To further enhance the gameplay experience, we have devised to remove the cards from the game board when they are chosen correctly by the player. This idea involves implementing a function that will be triggered when a pair of matching cards is found.

Implementation

One of our main functions is "setUpCards()", which is responsible for setting up the cards and images on the game board. The function begins by generating a matrix of the given size, which represents the positions of the cards on the game board. The positions of the paired cards are then randomly shuffled using the randperm function. These shuffled positions are subsequently reshaped to form a matrix of the specified size.

Next, a nested for loop is utilized to create the images and buttons for each card on the game board. The loop is initiated from the top left corner and iterates through each row and column, enabling the first cards to be referenced using the row and column numbers, similar to a vector. For each card, the corresponding image source is retrieved from the memory folder. Initially, the visibility of the image is set to "off" to hide the edges of the images behind the card backs (buttons).

The positions of the buttons are set to match the positions of the images on the game board. Additionally, each button is assigned a "ButtonPushedFcn", which links it to the "cardClicked()" function.

The "buttonPressTimerStart()" function ensures that the timer starts when a button is pressed, as long as the game has not already been won. It allows for time-based events and actions to be executed within the game, providing a dynamic and time-sensitive gameplay experience. If the

game has not been won, the function calls the `tic()` function to start the timer by getting the current time as the starting point for measuring the elapsed time. After starting the timer, the function then calls the `"countdown()"` function.

The `"countdown()"` function is an essential component of the game that manages the countdown timer and handles various game outcomes. This function is responsible for updating the display with the remaining time, checking if the countdown has finished, and determining whether the player has won or lost the game. At the beginning of the function, the countdown duration is set based on the selected difficulty level. The duration is stored in the `duration` variable, which determines how long the countdown will last.

The function then enters a loop that continues until a break condition is met. Within the loop, the remaining time is calculated by subtracting the elapsed time from the duration. The elapsed time is stored in the `elapsed_time` variable and is incremented by one second after each iteration. To provide a clear representation of the remaining time, the function converts the remaining seconds into minutes and seconds. The minutes are obtained by dividing the remaining seconds by 60 and taking the floor value, while the seconds are calculated using the `rem()` function.

The function checks two conditions within the loop to determine the game outcome. Firstly, it checks if the countdown has finished and the player has not found all the required pairs (`remaining_seconds <= 0 && found_pairs ~= 10`). If this condition is met, the function saves the result of the game, displays a losing message using a message box (`msgbox()`), and breaks out of the loop.

Secondly, the function checks if the player has found all the required pairs (`found_pairs == 10`). If this condition is satisfied, the function saves the result of the game, displays a winning message using a message box, and breaks out of the loop. To ensure the display is updated every second, a one-second pause is included using the `pause(1)` function. Once the countdown is complete, the display is updated to show "Countdown complete" by calling the `updateDisplay()` function, indicating that the timer has finished.

The `"updateDisplay()"` function is responsible for updating the user interface with the remaining time during the game. It first checks if the app object is valid and not empty to ensure safe access to its properties. If the app object is valid, it updates a UI element, such as a label, with the remaining time. Finally, `"saveResultOfGame(app)"` function is used to save important game information for the main game in a MAT file.

Our next function, `"cardClicked()"`, ensures that only two cards can be flipped over at a time by performing a safety check at the beginning of the function. If `counter_cards_visible` (which keeps track of the number of currently visible cards) is already equal to or greater than 2, the function returns without performing any further actions. When a card is clicked, the visibility of the corresponding button is set to "off" and to "on" for the hidden image behind the card to reveal.

After revealing the image, the function updates the `counter_cards_visible` variable by adding itself by 1. This variable keeps track of the number of currently visible cards. If `counter_cards_visible` is equal to 1, it means that the first card has been clicked. Consequently, the row and column of the clicked card are stored in `clicked_row(1)` and `clicked_col(1)` respectively. If `counter_cards_visible` is equal to 2, it indicates that the second card has been clicked. In this case,

the row and column of the second clicked card are stored in `clicked_row(2)` and `clicked_col(2)` respectively.

When `counter_cards_visible` reaches 2, indicating that two cards are now visible, the function introduces a pause of 1 second. For this the timer command is used to implement the pause between flipping two cards without blocking the main thread, so the countdown will not be influenced. This pause allows the player to view the second card before proceeding.

The "`compareCards()`" function plays a important role in the game by comparing the images on two selected card. First, the function checks if the images on the selected cards are the same. It does this by comparing the elements in the `images_numbers` matrix at the specified row and column indices. If the images match, it means a pair has been found. The matched cards are removed from the game. This is typically achieved by changing the visibility property of the associated image objects to hide them from view.

After removing the cards, the function checks if all pairs have been found. It does this by comparing the value of `found_pairs` (the number of pairs found) with the total number of pairs in the game. If all pairs have been found, indicating that the game is won, to records the current duration of the game in the `elapsed_time` variable. The specific method used to measure time may vary depending on the game implementation. The `game_won` flag is set to true, indicating that the game has been successfully completed. It assigns a result memory value of 1 to `result_memory`, which likely stores information about the game result. The function saves the game result using the "`saveResultOfGame()`" function, which typically writes important game information to a file.

On the other hand, if the images on the selected cards do not match, indicating that a pair was not found. The function restores the visibility of the flipped cards so that they are visible again. Finally, the function resets the count of visible cards to 0.

The "`removeCards()`" function is responsible for removing the matched pair of cards from the game board. By setting the visibility of the corresponding images to "off", the function effectively eliminates the cards from the game board, signifying a successful match.

The only callback function utilized in this context is "`startupFcn()`". This function is executed when the app starts up and is responsible for configuring the initial state of the app. It performs tasks such as adding a path and invoking the "`setUpCards()`" function to set up the game board.

User manual

- To start the game, click on any card to reveal the hidden image behind it.
- Remember the position and image of the card you just flipped.
- Now, click on another card to reveal its hidden image.
- If the two cards have the same image, it means that player found a matching pair.
- If the images on the two cards do not match, take note of their positions and images.
- The selected cards will automatically flip back over, hiding the images again.
- Continue flipping cards and trying to find matching pairs until all pairs have been discovered.
- Click the "Info" button for additional game instructions.

Challenges

Initially, it was necessary to ensure that the order of the cards and corresponding images in the for-loop matched the desired layout on the game board. Specifically, the cards needed to start from the upper-left corner rather than the lower-left corner. To achieve this, the indices of the cards were aligned with the indices of a matrix, with the row index being prioritized over the column index.

Another challenge involved accurately keeping track of the number of cards that were flipped over during gameplay. It was important to accurately determine when the maximum allowed number of cards was reached, as this would impact the logic and flow of the game.

To facilitate the comparison and removal of matching pairs, it was necessary to save the row and column numbers of the two selected cards. However, a challenge accrued in finding a way to store this information without overwriting the numbers of the first selected card. A vector data structure was employed to preserve and retrieve the row and column numbers of the selected cards separately.

One of the challenges we faced was that the buttons we used only had an icon function, which limited the visibility and size of the associated image. To address this, a new idea was conceived. The images were hidden behind the buttons, making the buttons themselves represent the back of the cards. This allowed the images to be displayed more clearly and prominently to the players.

One of our toughest challenges revolved around dealing with the countdown timer and its related functions. Figuring out when to start the timer, when to finish it, and when to display the winning or losing message box was quite tricky. However, the most challenging part was figuring out how to stop the timer when the player won the game earlier than expected.

Sudoku

This sudoku 9x9 game offers a grid of numbers that aims to test your logical thinking and problem-solving abilities. Your task is to fill in the empty cells with numbers from 1 to 9, ensuring that each row, column, and 3x3 sub-grid contains all the numbers without any repetition. It is crucial to carefully consider the provided clues and rely on deductive reasoning to achieve success.

What makes this "Sudoku" mini game unique is its one-time play feature. The reason behind this lies in our main game, the Snake and Ladder. Within this game, players are presented with an intriguing decision: whether to take a risk. Choosing the "Risk button" may result in encountering the Sudoku mini game as a challenge. If players win in this challenge, they will receive corresponding rewards based on the chosen difficulty level.

Fundamental ideas

In the Sudoku game, the first step is to provide players with a grid where numbers can be entered. To achieve this, an "EditField" feature is incorporated, specifically designed to accept numeric inputs. Since a 9x9 Sudoku grid is aimed to be created, a total of 81 fields are needed.

Once the grid is set up, the next aspect is defining the default numbers based on the chosen difficulty level. Different difficulty levels present varying degrees of initial complexity by pre-filling certain numbers in the grid. These default numbers serve as starting points for players and establish the initial puzzle state. Initially, the idea of using the rand() function to generate random default numbers displayed in random fields was considered. However, instead, three pre-defined "Sudoku" samples are created and represented as matrices.

To determine whether a player has successfully completed the game, a mechanism to check their answers needs to be implemented. For this purpose, the consideration of incorporating a button was made. After players have filled in the entire grid, their solution will be verified against the Sudoku rules. If the Sudoku rules are satisfied by their numbers (no repeated numbers in rows, columns, or subgrids), they will be declared the winners of the game.

Implementation

We have two main functions that are required for this mini game: "generateSudoku()" and "checkSudoku()".

The first essential function is "generateSudoku()". This function takes the difficulty level as input and generates the Sudoku game accordingly. Within this function, a matrix is used to define the default numbers and their corresponding fields. By utilizing for loops, the default numbers are displayed on the game board. It is important to note that the default numbers shown in the grey fields upon clicking the "Generate Sudoku" button will differ based on the chosen difficulty level.

The second function, "checkSudoku()", is responsible for evaluating the player's answers at the end of the game and determining whether they have won or lost. The input for this function is the difficulty level. Within this function, the correct answers are defined as a matrix. Using a for loop, the player's entered values are compared to the correct answers. If a player's answer matches the corresponding correct answer, the field will change color to green; otherwise, it will turn red. After the for loop, an if statement is used to check if all answers are correct. If they are, a message box displaying " You won. Please close the minigame to continue the main game and roll the dice once again to get your reward" will appear. Otherwise, the message " You lose! No rolling dice for you. Please close the minigame and click on your figure to finish your previous move." will be displayed. In both cases, the important game information is saved with the "saveResultOfGame()" function.

For this mini game, we require three callbacks. The first one is the "startupFcn()", which is automatically triggered when the app is launched. Its purpose is to perform any necessary setup or initialization tasks before displaying the app interface to the user.

In the "startupFcn()", a uiaxes element is added to create two vertical and two horizontal axes that separate the sub-grids. To achieve this, the x and y limits of the uiaxes are defined, and they are plotted using the xline and yline functions. Afterwards, the axes are made invisible to the user.

Another callback we require is "GenerateAnswersButtonPushed()". This callback is triggered when the player clicks the "Generate Sudoku" button. Within this callback, the "generateSudoku()" function needs to be called with the input of the selected difficulty level. By doing so, the "generateSudoku()" function will generate the default numbers for the Sudoku game based on the chosen difficulty level that the player selected at the beginning.

The next callback we need is "CheckAnswersButtonPushed()". This callback is triggered when the player clicks the "Check Answers" button. Within this callback, the "checkSudoku" function simply needs to be called with the input of the difficulty level. By doing so, the "checkSudoku" function will check all the player's answers according to the chosen difficulty level that the player selected at the beginning.

The "UIFigureCloseRequest()" callback function is utilized to handle the event triggered when the user attempts to close the user interface (UI) figure associated with the mini-game. This callback function ensures that the main game is notified when the mini-game is closed, enabling the main game to resume its execution.

User manual

- To start the game, click on the "Generate Sudoku" button. The grid will automatically be filled with default numbers, varying according to the chosen difficulty level. These default numbers will be shown in the grey cells.
- Select any empty cell within the grid and input a number from 1 to 9 using either the keyboard or number pad. Remember, the goal is to fill the grid so that each row, column, and 3x3 sub-grid contains all numbers from 1 to 9 without repetition.
- Since this mini-game only can be played once, it's important to be confident in your answers. Once all the grids are filled, click on the "Check Answers" button.
- If all the entered numbers are correct, players will receive a congratulatory message stating, depending on the chosen difficulty level, they will be rewarded accordingly. Otherwise, they lose the game and they will be notified with a message stating.

Challenges

One of the biggest challenges we faced while coding the Sudoku game was figuring out how to set the default numbers for the grid. Initially, we thought about using the "rand" function to randomly assign numbers to different fields. But then we ran into trouble when we tried to write a function to check if the player's answers were valid. Instead of relying on random numbers, we decided to use three pre-made "Sudoku" puzzles for the easy, medium, and hard levels. We stored these puzzles as matrices, which made it easier for us to know the correct answers. Now, we could compare the player's answers with the predetermined solutions using our "checkAnswer()" function. It made the whole process much simpler and more reliable.

Future Development

As part of our plans for further development, we considered implementing several features to enhance the user experience. One idea was to include a high score list button within the main game app, allowing players to easily check their scores. Additionally, we thought about incorporating a dark/light mode option to provide different user preferences. We also considered adding language selection functionality to make the game accessible to a wider audience. In terms of audio, we planned to include a sound on/off toggle, giving players control over the game's audio experience. Lastly, we wanted to address color blindness concerns by providing options specifically designed for players with color vision deficiencies. These proposed developments aimed to improve the overall gameplay experience and ensure inclusivity for a diverse range of users.

Conclusion

In conclusion, our project encountered various challenges and difficulties that required us to maintain determination and think creatively. We faced obstacles such as determining the placement of figures, handling situations when two figures occupied the same spot, and addressing missing functions in the table.

However, we did not allow these challenges to overcome us. Instead, we checked all the codes multiple times, comprehended its workings, and found intelligent solutions. We learned to approach problems with patience and persistence, finding alternative approaches to circumvent limitations or lack of features. These skills will undoubtedly prove beneficial for future projects.

During the project, we made an effort to explore different coding approaches, from smart and efficient techniques to less optimal ones. This helped us gain insights into various coding styles and methods, allowing us to develop and enhance our coding skills. We spent time researching and experimenting with different coding styles and algorithms, testing them to see what worked and what didn't. This hands-on approach gave us a better understanding of the strengths and weaknesses of different coding methods. This collaborative process helped us evaluate our code critically, identify areas where we could improve, and aim for more elegant and efficient solutions.

In addition to our coding efforts, we also faced challenges in collaborating and sharing our files effectively. However, we overcame this hurdle by utilizing GitHub to create a new repository and integrate it into our MATLAB workflow. Although coordinating simultaneous work posed difficulties at times, we successfully managed to collaborate.

Despite the obstacles, we managed to make progress and achieve our goals. We successfully implemented the necessary adjustments and attained the desired outcomes. As a team, we have a strong sense of satisfaction in the solutions we formulated and the progress we achieved. This project served as a valuable learning experience, highlighting the significance of perseverance, critical thinking, and teamwork when confronted with challenges.

Declaration of Authenticity

We, the undersigned members of the team, hereby declare that the computer programming project titled "[SnakeAndLadderGame]" submitted by us is the result of our collective efforts and collaboration. We affirm that all code, algorithms, designs, and concepts presented within this project are original and have been created by us, jointly and individually, unless otherwise cited.

We further declare that all resources, including but not limited to libraries, frameworks, and external code snippets, utilized in the development of this project have been appropriately credited and referenced in accordance with academic integrity standards.

Yeganeh Mohammadi

Sabrina Miller

Giessen, 10.03.2024