Alice in Wonderland Board Game Documentation

Welcome to the documentation for the Alice in Wonderland Board Game! This whimsical project brings Lewis Carroll's enchanting world to life, blending creativity, fun, and a sprinkle of adventure into an interactive board game. Here's a roadmap of what to expect:

Roadmap of the Report

1. **Introduction**
   An overview of the project's goals and how this documentation is structured to guide you through its development.
2. **Background**
   A deep dive into the game's concept, rules, and the logical framework that ties everything together.
3. **Specifications and Design**
   A closer look at the features that bring the game to life, including functional and non-functional requirements, design principles, and system architecture.
4. **Implementation and Execution**
   The journey of building the game: tools, techniques, creative solutions, and challenges tackled along the way.
5. **Testing and Evaluation**
   How we made sure the game was fun, reliable, and polished. This includes testing methods, insights from evaluation, and any limitations.
6. **Conclusion**
   A summary of what we achieved and the exciting possibilities for future updates and enhancements.

INTRODUCTION

Aims and Objectives of the Project

The project aims to create a kid-friendly, interactive board game inspired by Alice in Wonderland, combining fun and strategic gameplay. Built on the mechanics of Snakes and Ladders, the game incorporates thematic elements like rabbit holes (snakes), collectible tea cups, and surprise twists to enhance player engagement.

Key objectives include:

- Interactive Gameplay: Designing an engaging game board with Alice in Wonderland–themed elements and beloved characters like Alice and the Cheshire Cat.
- Dynamic Features: Introducing mechanics such as special squares, item collection, and bonus rolls to add layers of strategy and surprise to the experience.
- Database Integration: Seamlessly track gameplay progress, including cup collections, ladders climbed, and rabbit holes fallen into, to provide a personalized and engaging player experience.

BACKGROUND

Project Concept, Gameplay and Logic

There are two characters/players, Alice and the Chesire Cat, who roll the dice to move across the board, encountering rabbit holes that set you back and ladders that help you advance. The tea cup squares grant extra rolls when you land on them. The game ends when a player reaches the final square. The game is automated, with progress and key data tracked in a database, including tea cup collections, time, and special squares..

Game Features

- Automated Gameplay: All mechanics, including dice rolls and movements for both Alice and the Cheshire Cat, are automated by pressing 'Enter' to roll the dice and move accordingly.
- Pop-Up Feedback: At the start, the main menu appears with options to play, view directions, or quit. Throughout the game, if you land on a special square, a pop up message with appear.After the game ends, a result screen allows you to play again, review directions, or exit.
- Event Messages: The sidebar shows your dice roll.

SPECIFICATIONS AND DESIGN:

Core Mechanics

- Movement: You roll a dice to move Alice across the board.
- Special Squares: Rabbit holes pull Alice back to the previous rabbit hole, while ladders advance her forward.
- Tea Cups: Landing on a tea cup gives the player an extra roll.
- Game Completion: The game ends when a player reaches the final square, completing the game.

Database Integration

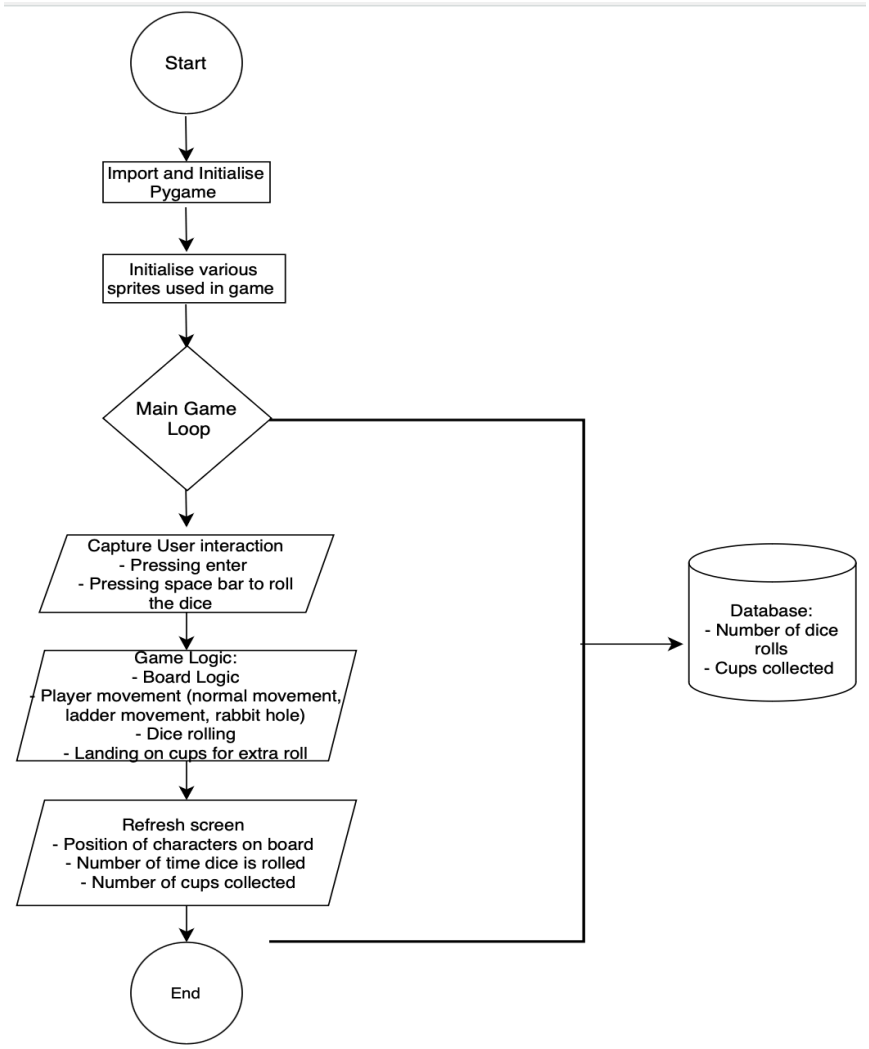
- Tracking Data: The database stores progress, including item collections, ladders climbed, and rabbit holes fallen into.

Functional vs. Non-Functional Requirements

- Functional Requirements:
  Dice rolling, board interactions, database tracking.

- Non-Functional Requirements:
  Themed aesthetics, real-time feedback, and reliable performance.
  - Comment: Currently, it is a two-player game. Real-time feedback provides updates on collectables obtained by both you and your opponent.. Reliable performance ensures all these features run smoothly without crashes.

FLOW DIAGRAM:



Step-by-Step Gameplay Flow

Step 1: Starting the Game

- What the Player Sees:
    - When the game launches, a pop-up menu appears with options to Play, Directions, or Quit.Pressing enter takes you to the board.
    - If you select Directions, a list of game mechanics and instructions on how to play will appear. You can return to the main menu by pressing the back button. Additionally, the X button in the top right corner can be used to exit the game and quit Pygame.
- What Happens Behind the Scenes:
    - Running main game loop - The database initializes set up to track your teacup collection and special squares for the ladders and rabbit holes.

    - The board and side panel are set up, ready to display dice rolls, movement summaries, and collected teacup counts.

Step 2: Rolling the Dice

- What the Player Does:
  Press the space bar to roll the dice and move your character across the board.
- What Happens Behind the Scenes:
    - The dice roll determines how many spaces the player moves.
    - The system updates their position, checks the type of square they landed on, and triggers any associated events (e.g., sliding down a rabbit hole, climbing a ladder, or getting an extra roll).

Step 3: Interacting with the Board

Landing on a Rabbit Hole:

- What You See:
  The character will appear in the new position below, at the exit of the rabbit hole.
- What Happens Behind the Scenes:
    - The system adjusts Alice's position on the board and logs the event.

Landing on a Ladder:

- What You See:
  The character will appear in the new position above, at the top of the lader.
- What Happens Behind the Scenes:
    - Alice’s position is updated, and the system continues tracking progress.

Landing on a Teacup Square:

- What You See:
  A message says: "Alice has landed on a tea cup! You have an extra roll!"
- What Happens Behind the Scenes:
    - The teacup count increases in the database.

Step 4: Game Continuation

- What You See:
  The game alternates between Alice and the Cheshire Cat. The side panel updates after every roll, showing:
    - Dice results.
- What Happens Behind the Scenes:
    - The system ensures smooth transitions between turns by clicking the ‘Enter’ button and keeps all data synchronised in the database.

Step 5: Game Conclusion

- What You See:
    - If Alice reaches the final square first: "Congratulations! Alice escaped Wonderland! You win!"
    - If the Cheshire Cat wins: "The Cheshire Cat outpaced Alice! Better luck next time."
    - A pop-up menu appears with options to Play Again, Directions, or Quit.
- What Happens Behind the Scenes:
    - The system checks after every turn to see if either Alice or the Cheshire Cat has reached the final square.
    - Once the game ends, the final scores are saved in the database, and the message is displayed.

IMPLEMENTATION AND EXECUTION

Development Approach and Team Roles

Creating our board game was a collaborative effort. We divided tasks based on our strengths, ensuring everyone contributed to a specific area while maintaining a shared vision for the final game.

Team Contributions:

- Players (Muna): Focused on building the player objects, ensuring their positions and teacup counts updated dynamically during gameplay.
- Movement Triggers (Agnes): Developed the dice roll logic and handled interactions between the board and players, like climbing ladders or falling into rabbit holes.
- Board Mechanics (Naomi): Designed the game board layout, integrating rabbit holes, ladders, and teacup squares with clear, functional rules.
- Dice (Vicki): Programmed the dice-rolling mechanic to ensure smooth and randomised outcomes.
- Graphics (Sofia): Designed and implemented visual elements to maintain the Alice in Wonderland theme, ensuring the visual elements was functional and visually engaging. Created the popups for entering the main menu of the game and exit menu. Documentation of project.
- Database (Hannah): Set up the database to track player stats, like teacup collections and game duration.

Tools and Libraries

- Libraries:
    - PyGame: Powered the game's graphics and core mechanics.
    - SQL: Stored player data such as teacup collections and scores.
    - Random and os: Managed dice rolls and file operations.
- Tools:
    - GitHub: Used for version control and collaboration.
    - Slack: Kept us connected for regular updates and check-ins.

- Trello: Used to keep track of ticket items and how each task was progressing

Implementation Process

1. Planning and Brainstorming:
   We began with brainstorming and chose the Alice in Wonderland theme, incorporating elements of "Snakes and Ladders." Tasks were divided based on individual expertise and personal interests, ensuring the project remained manageable and well-organised.
2. Development Stages:
   - Core Mechanics: We focused on coding the dice rolls, player movement, and interactions with special board spaces like rabbit holes, ladders, and teacup squares.
   - Graphics: We brought the theme to life with carefully chosen visuals and adjusted the layouts to make sure everything—from the sidebar and dice animations to the graphical rolls and movement across the board—worked together smoothly and looked polished.
   - Database Integration: Configured to store data like teacup collections and recorded special square scores to make player progress trackable and persistent.
1. Collaboration:
   - GitHub allowed us to work asynchronously, review each other's work, and track progress.
   - Regular Slack updates ensured we were aligned and provided opportunities to ask for help or share feedback.

Challenges and Solutions

1. Time Management: Coordinating schedules across the team was a bit tricky, but assigning tasks that could be worked on individually really helped, especially given everyone's different time capabilities. Using GitHub and Slack kept everyone updated on where we were and ensured steady progress. We also stepped in to help each other when needed, which helped maintain a smooth workflow.
2. Component Integration:
   Combining the dice mechanics, board interactions, and database syncing required multiple iterations and debugging. It took some time to get everyone's components working together, especially when individual parts were ready for testing at different times, but we managed to work through it with trial and error. Team reviews helped identify and resolve compatibility issues along the way.
3. Graphics Adjustments:
   Ensuring our visuals worked within PyGame's framework involved some trial and error. We consistently refined asset sizes and placements to make sure the sidebar remained clean and engaging, displaying pop-ups, dice rolls, player updates, and event notifications smoothly. We also focused on organising the asset folders and file structures, keeping everything in sync to maintain a clean project as we made updates.

Agile Development:

- We focused on implementing core features, like movement and dice rolls, first to ensure the game was playable. Once those were stable, we added extra elements, such as teacup rewards. Regular code reviews helped us refine and optimise the

gameplay mechanics. We also remained adaptable, adjusting our plans as necessary to ensure a solid foundation before refining the design or adding additional features.

TESTING AND EVALUATION

Testing Strategy

Although formal user testing hasn't been conducted, we've focused on ensuring the game's core mechanics are both functional and engaging through simulation-based testing.

Functional Testing

We ran a series of unit tests to verify all major components are working as intended. These tests focused on key mechanics such as:

- Dice Roll Simulation: Ensured dice rolls were fair and consistent.
- Player Movement: Tested standard movement mechanics, ladder interactions, and rabbit hole mechanics.
- Cup Bonuses: Verified bonus opportunities triggered correctly when landing on specific squares.
- Position Logic: Checked that logical player positions matched their on-screen counterparts.
- Edge Case Handling: Simulated invalid moves to ensure graceful error handling.

These tests confirmed that the game mechanics are functioning smoothly and remain simple enough to maintain a fun, intuitive experience.

System Limitations and Opportunities for Improvement:

While the game runs as expected, there are opportunities to expand its features and polish its design:

1. Multiplayer Functionality:
   The game currently supports a single-player experience against an automated 'Cheshire Cat' opponent. However, introducing multiplayer would elevate the experience by adding a social, competitive element. Possible directions for multiplayer include:
   - Peer-to-Peer Multiplayer: Retaining the classic social board-game feel by allowing two players to share a game locally on one device or multiple devices..
   - Server-Based Logic: Expanding into a web-based multiplayer option to ensure accessibility, while always maintaining the option for a single-player experience.
2. Visual Movement and Animations:
   Currently, Alice's movement is instant—she teleports directly to her destination after rolling the dice. While this choice adds thematic charm and aligns with the Cheshire Cat's

whimsical nature, introducing visual movement animation could make her movement smoother, more visually appealing, and more immersive.

- Animations could balance Alice's human-like movement with the game's overall magical and thematic style.
- Keeping elements of sudden teleportaion movement that was purposefully implemented to imitate the disappearing act of the 'Cheshire Cat' for the opponent could maintain the game's charm while creating an interesting development.

Future Evaluation Strategies

To further refine the game mechanics and ensure a smooth player experience, the following strategies will be prioritised:

1. More Edge Case Testing:
   Ensuring ultimate smoothness means more testing various edge cases, such as:
   - Rolling the same dice number consecutively.
   - Landing in bonus areas or triggering special mechanics.
   - Movement overflow—e.g., overshooting the winning condition (e.g., moving past position 30 to 32)—and testing how the game logic handles these scenarios.
2. Unmapped Ladders & Rabbit Holes Testing:
   Testing invalid ladder or rabbit holes and logic errors will ensure the game responds predictably when unexpected behaviour occurs.
3. Input Validation:
   Edge-cases to test invalid user inputs—such as incorrect dice numbers, invalid clicks, or unexpected interactions—to ensure that they don't cause crashes in teh system.
4. UI-Related Testing:
   While UI mechanics have functioned well so far, future testing should target:
   - Testing visual effects, dice roll feedback and roll scores on screen for both contestants.

These testing strategies aim to create a smooth, engaging player experience while addressing potential mechanical gaps and user interactions.

CONCLUSION

We've crafted a game that blends simple mechanics with playful elements like rabbit holes and teacups, offering an experience that's both intuitive to explore and full of unexpected charm.

The most exciting part of this journey was the creative challenge of building an original, fun game we truly believed players would enjoy exploring. Working as a team to bring different aspects of the game to life was both rewarding anda great learning experience. While teamwork came with its challenges—especially as each of us worked on different mechanics at different stages—it was successful thanks to clear communication and a shared commitment to the project.

One key lesson was focusing on keeping the game's core simple by prioritizing core mechanics first and building outward, adding bigger features as the game developed to meet time

constraints and ensure completion. This was our first time using Pygame, and it was fascinating to see how our code visually came to life. Translating technical elements into interactive visuals was both exciting and educational.

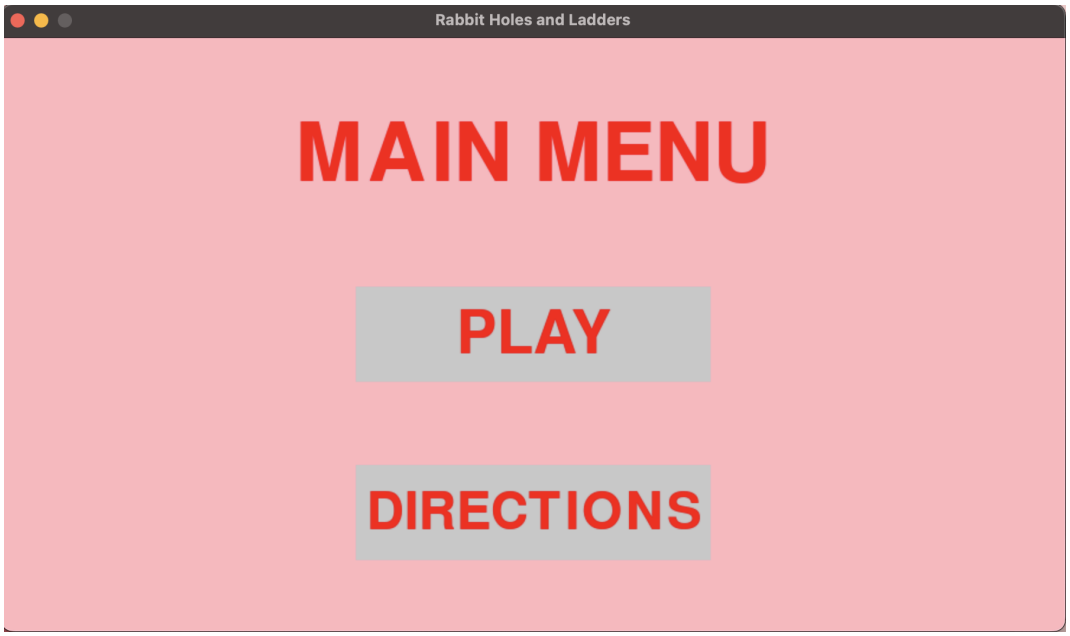
We're particularly proud of the small but transformative design changes, such as the teacup offering extra rolls and replacing snakes with rabbit holes. These choices added unpredictability and charm while maintaining the *Alice in Wonderland* theme.

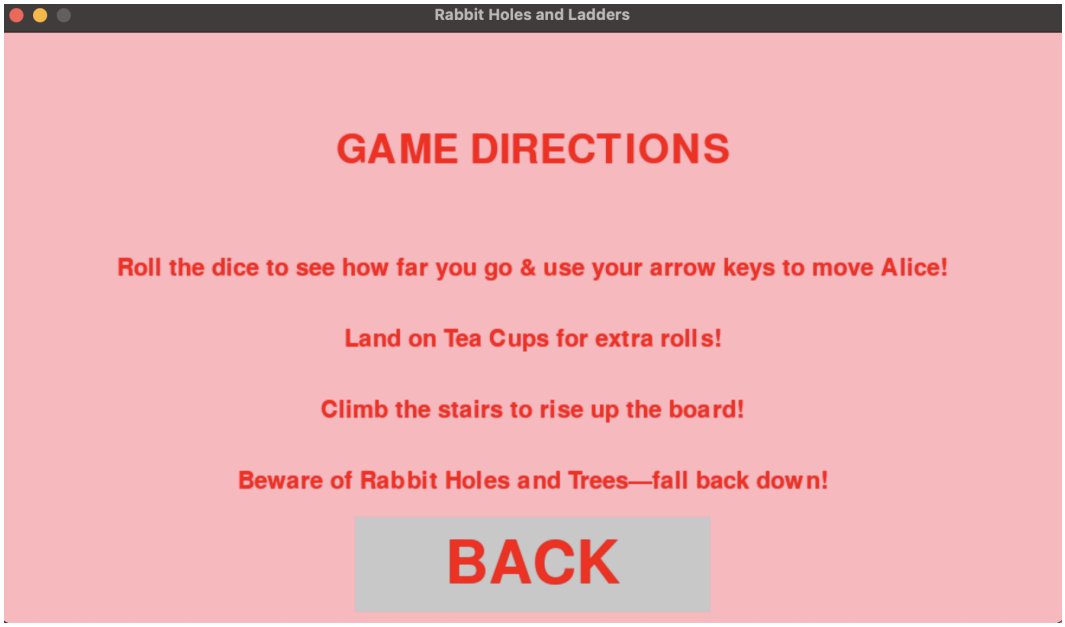
While the game is fully functional and fun as it stands, we're inspired by the potential for future improvements. Possibilities include adding multiplayer modes, letting players select customizable themes, or implementing new mechanics like puzzles or mini-challenges. We also explored allowing players to adopt unique Wonderland-inspired characters with their own quirks and abilities. Interactive visuals—like pop-ups featuring the Caterpillar sharing fun facts or riddles—could further enhance the immersive experience.

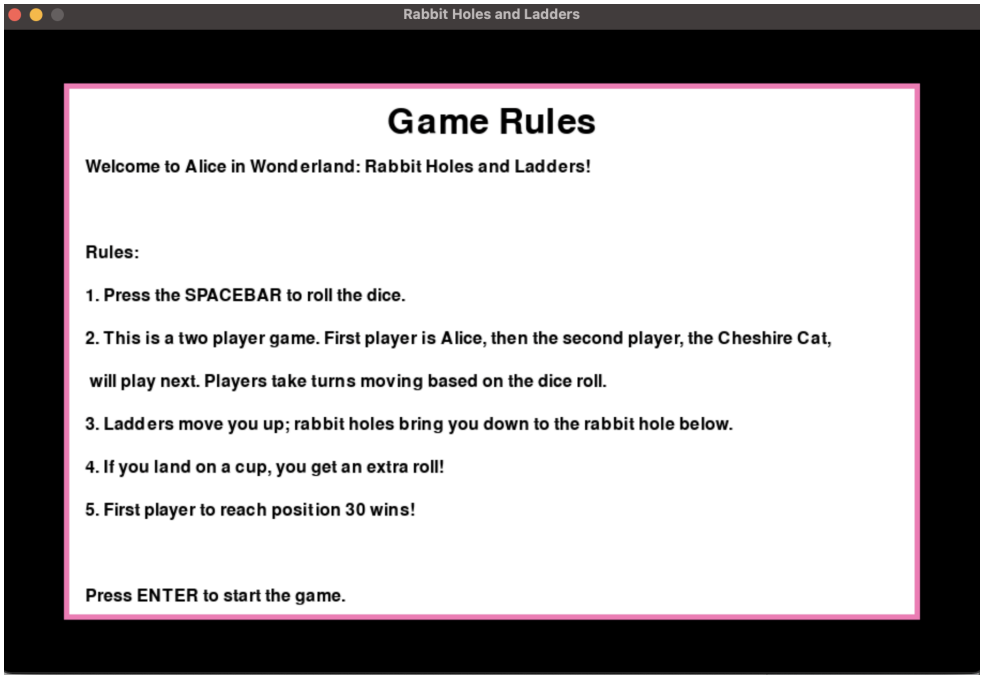
We did have plans for a main menu pop-up to enhance navigation when first running the game but ran out of time to implement it fully. (Please see below for the intended opening, which is coded but wasn't fully integrated due to time constraints and technical probelms. This was replaced by a simplified version of the code, that offers directions and an "Enter" option.) Instead, we focused on keeping the core mechanics simple, emphasising clear instructions so players could jump right into the fun.

At the end of the day, this *Alice in Wonderland Board Game* is a mix of technical exploration, creativity, and teamwork. Building it was more than just executing code—it was about learning, problem-solving, and collaboration. We're proud of what we've built and are excited about what could come next.

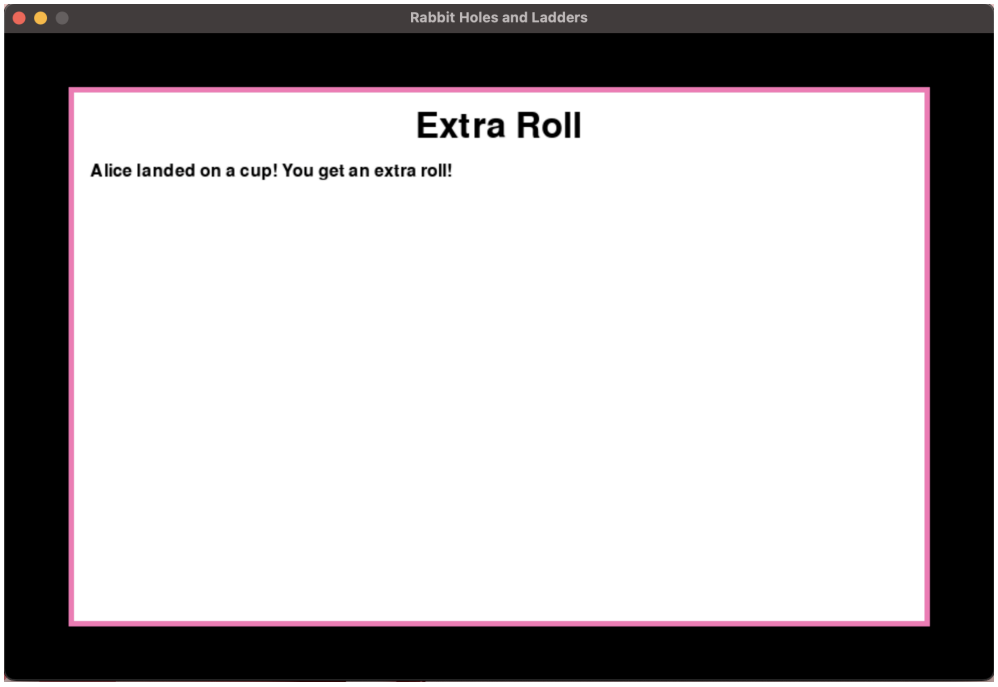
GAMES VISUAL LOGISTICS

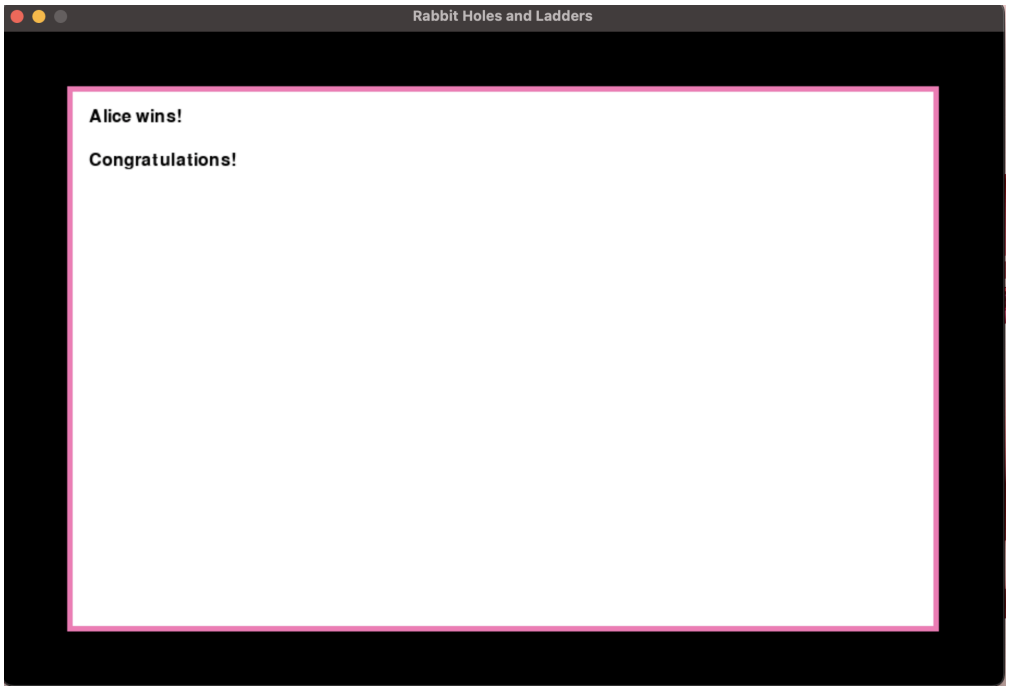





Rabbit Holes and Ladders
Game Rules
Welcome to Alice in Wonderland: Rabbit Holes and Ladders!
Rules:
1. Press the SPACEBAR to roll the dice.
2. This is a two player game. First player is Alice, then the second player, the Cheshire Cat,
will play next. Players take turns moving based on the dice roll.
3. Ladders move you up; rabbit holes bring you down to the rabbit hole below.
4. If you land on a cup, you get an extra roll!
5. First player to reach position 30 wins!
Press ENTER to start the game.


Rabbit Holes and Ladders
25
26
27
28
30
24
22
21
20
19
13
14
15
16
17
18
12
11
10
9
8
7
1
2
3
4
5
6
Alice rolled: 2
Cat rolled: 4


Rabbit Holes and Ladders
Extra Roll
Alice landed on a cup! You get an extra roll!


Rabbit Holes and Ladders
Alice wins!
Congratulations!

CRITERIA CHECKLIST (WITH SCREENSHOTS):

1.EFFECTIVE CLASS, METHOS AND VARIABKE NAMES

We made sure that our class, method, and variable names were meaningful and self-explanatory. Names like Player, move_player(), and ladders_and_snakes clearly describe their purpose and function. This made the code intuitive to read and understand.

Demonstartion:
For example, the Player class represents each player in the game, and its methods, like move_player(), handle specific tasks like movement across the board. Here's a snippet:

```
181  class Player(pygame.sprite.Sprite):  4 usages
182      def __init__(self, id, image):  NayMcE
183          super().__init__()
184          self.id = id
185          self.image = image
186          self.rect = self.image.get_rect()
187          self.position = 1
188
```

```
def move(self,steps):  6 usages
    self.position += steps
    if self.position > 30: #ma
        self.position = 30
```

```
ladders_and_rabbit_holes = {
    28: 17, #rabbit_hole
    17: 12, #rabbit_hole
    12: 1,  #rabbit_hole
    4: 10, #ladder
    11: 22, #ladder
    18: 29 #ladder
}

cup_positions = [2, 7, 24]
```

2. EFFECTIVE TOP-DOWN DECOMPISITION OF ALGORITHIMS

We broke down the logic into small, manageable functions to keep the code clean and maintainable. Each function performs a single, focused task. This approach made debugging and updates much easier.

Demonstartion:

For instance, functions like roll_dice() handle dice rolls, while check_ladders_or_snakes() checks if a player encounters a ladder or snake. This division of tasks keeps our routines cohesive and readable:

```
def move(self,steps):  6 usages  NayMcE
    self.position += steps
    if self.position > 30: #maximise the size of the screen
        self.position = 30

    if self.position in ladders_and_rabbit_holes:
        self.position = ladders_and_rabbit_holes[self.position]
```

3. CODE LAYOUT SHOULD BE READABLE AND CONSISTENT

We followed good practices for formatting, such as using consistent indentation, spacing between blocks, and clear comments. This made the code clean and easy to read.

Demonstartion:

The code has a structured and uniform layout throughout. Here's an example of our formatting:

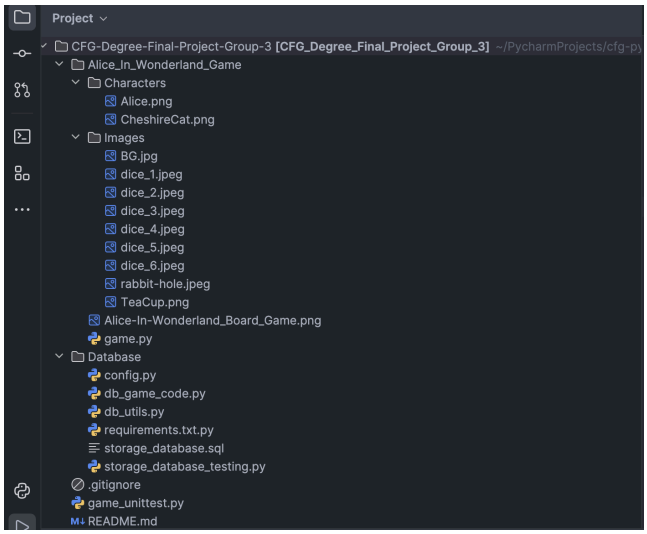
```
def roll_dice_animation():  2 usages  NayMcE
    roll_count = 10  # Number of frames in the roll animation
    for i in range(roll_count):
        # Display random dice face during the animation
        random_face = random.choice(dice_images)
        # Clear the dice animation area
        screen.fill(WHITE,  rect: (770, 250, DICE_SIZE, DICE_SIZE))
        screen.blit(random_face,  dest: (770,250)) # Animation position
        # draw_side_panel(3)  # Keep the side panel visible
        pygame.display.update()
        pygame.time.delay(50) # Short delay to simulate animation

    # Final roll result
    roll_result = random.randint( a: 1,  b: 6)
    screen.fill(WHITE,  rect: (770, 250, DICE_SIZE, DICE_SIZE))
    screen.blit(dice_images[roll_result - 1],  dest: (770, 250))
    # draw_side_panel(roll_result)  # Update side panel with final roll
    pygame.display.update()
    return roll_result
```

4. EFFECTIVE SOURCE TREE DIRECTORY STRUCTURE

We organized our files effectively by keeping related elements together and removing unused files. This ensures a clean and professional project structure.

Demonstartion:

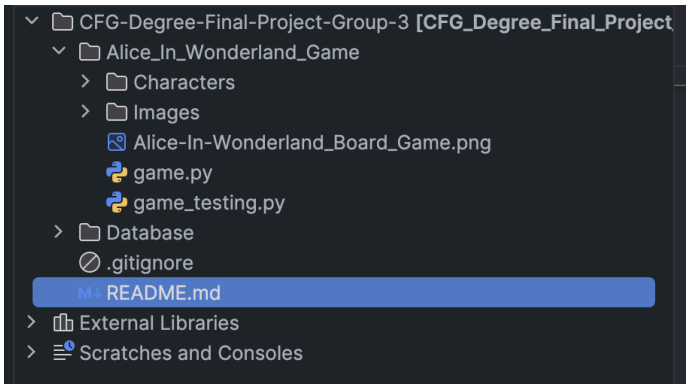
The source files were grouped logically, and we avoided clutter by removing unnecessary files.



5. EFFECTIVE FILE ORGANISTAION

We separated different aspects of the program into their own files where appropriate. This approach keeps the code modular and easier to navigate.

Demonstartion:
For instance, the game logic was kept in one file, while any supporting functions or assets (like images) were managed separately.



6. CORRECT EXCEOTION HANDLING

We added exception handling where it was needed to ensure the program could handle unexpected issues gracefully, like invalid inputs or file errors.

Demonstartion:
Here's an example of how we handled exceptions:

```
}
if position not in board_positions:
    raise ValueError(f"Invalid board position: {position}")
    # return board_positions[position]
```

This ensured that the program doesn't crash abruptly and provides meaningful feedback when something goes wrong.

7. GOOD UNIT TESTS

We wrote simple, targeted test cases to ensure that key functions in our code worked as expected. This gave us confidence that the game logic was solid.

Demonstartion:
For example, we tested the check_ladders_or_snakes() function with various inputs to verify it returned the correct position:

```
31
32  class TestPlayer(unittest.TestCase):  NayMcE
33      @classmethod  NayMcE
34      def setUpClass(cls):
35          # Initialize pygame to avoid errors when loading assets
36          pygame.init()
37          # Mock an image for player creation
38          cls.mock_image = pygame.Surface((100, 100))
39
40      def setUp(self):  NayMcE
41          # Create a Player instance for testing
42          self.player = Player( id: 1, self.mock_image)
43
44      def test_move_normal(self):  NayMcE
45          """Test normal movement (no ladder or rabbit hole)."""
46          self.player.move(3)  # Move 3 steps from position 1
47          self.assertEqual(self.player.position,  second: 4)
48
49      def test_move_with_ladder(self):  NayMcE
50          """Test movement where a ladder is present."""
51          self.player.position = 4  # Set initial position to 4
52          self.player.move(0)  # No steps, just test ladder logic
53          self.assertEqual(self.player.position,  second: 10)  # Ladder from 4 to 10
54
```

8. API

This project didn't require the use of an external API since it was focused on game logic that we created using Pygame.

9. OOP

We used OOP principles to structure the program. Classes like 'Player' encapsulated data and behavior, making the code modular and reusable.

Demonstartion:
The 'Player' class is a great example. It stores information about the player (like their name and position) and provides methods to manipulate that data. This use of OOP principles made the code more organized and scalable.Here's a snippet:

```
class GamePopup:  1 usage  NayMcE
    def create_popup(self, content, title=None):  3 usages  NayMcE
        # Add content (supports multiline)
        for line in content:
            split_lines = line.split('\n')  # Handle manual line breaks
            for sub_line in split_lines:
                text = self.font.render(sub_line, True, (0, 0, 0))
                popup.blit(text, dest: (20, y_offset))
                y_offset += 40

        return popup
```

10. Libraries

We used libraries like pygame, random, and sys to enhance the program. Each library served a specific purpose, from rendering graphics to generating random dice rolls.

Demonstartion:

- **pygame:** Used for graphics, event handling, and game interface.
- **random:** Enabled us to simulate dice rolls.
- **sys:** Helped us manage the program flow and exit cleanly when needed.

```
import pygame
import random
import sys
# Initialize pygame
pygame.init()
```