SCHOOL OF ENGINEERING AND TECHNOLOGY

FINAL ASSESSMENT FOR THE BSC (HONS) INFORMATION TECHNOLOGY; BSC (HONS) COMPUTER SCIENCE; BACHELOR of SOFTWARE ENGINEERING (HONS)YEAR 2

ACADEMIC SESSION 2024; SEMESTER 3

PRG2104: OBJECT ORIENTED PROGRAMMING

Project **DEADLINE: Week 14**

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 50% to your final grade.
- This is an individual assignment.

IMPORTANT

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

- Coursework submitted after the deadline will be awarded 0 marks
-

Lecturer's Remark (Use additional sheet if required)

I..... (Name) .....std. ID received the assignment and read the comments..... (Signature/date)

Academic Honesty Acknowledgement

"I .....Lim Jia Xuan.....(student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment."

…….. 21/8/2024..... (Student's signature / Date)

Contents

1.0 Introduction

The classic game of Rock-Paper-Scissors, though simple in nature, has long been a staple in decision-making and entertainment across cultures. The primary objective of this project is to bring this timeless game to life using ScalaFX, harnessing the power of modern programming paradigms to craft an engaging, interactive experience that goes beyond the basic mechanics of the game.

1.1 Rules of the Game

This is a player vs. computer game. At the start of the game, both sides receive 6 cards: 2 Rock Cards, 2 Scissor Cards, and 2 Paper Cards. As the game begins, the player selects a card, while the computer randomly chooses one. The cards are then compared, with the winner taking the loser's card. If it's a tie, both cards are removed. The card comparisons are as follows:

- Rock crushes Scissors
- Scissors cuts Paper
- Paper covers Rock

The game continues until one side has no remaining cards, at which point the side with remaining cards is declared the winner. Therefore, a key tip for this game is to remember and calculate the probability of the computer’s remaining cards and choose your cards accordingly.

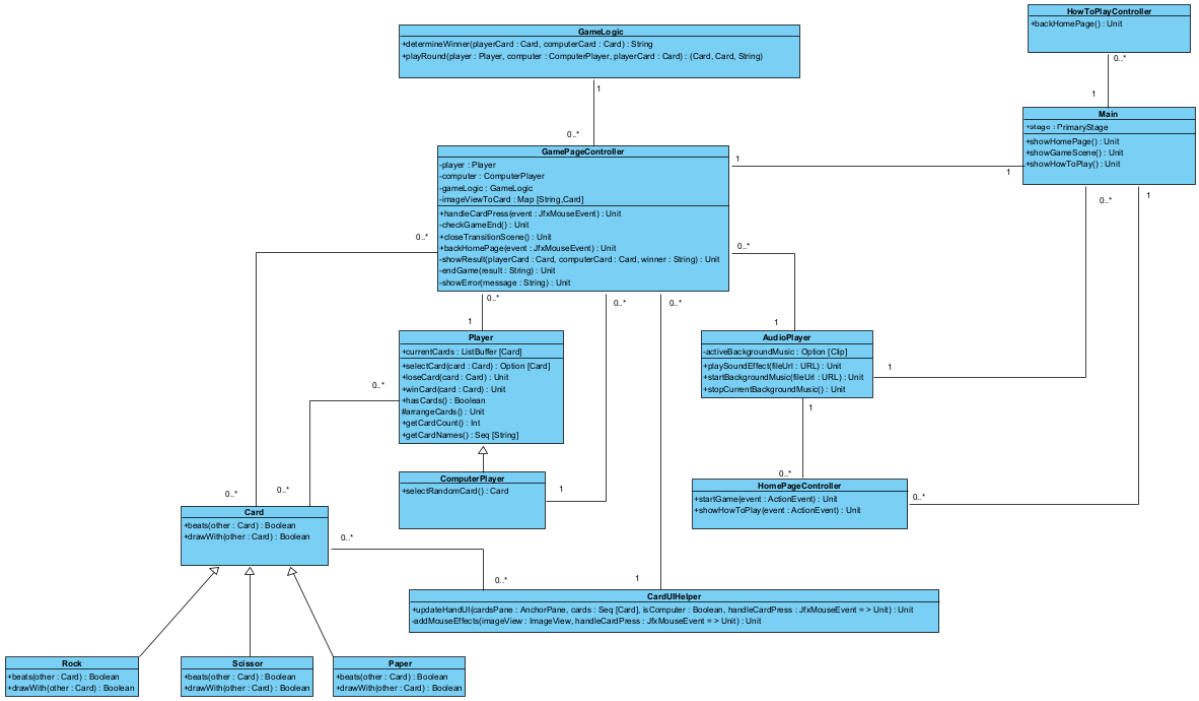
1.2 Motivation

The motivation behind this project is multifaceted, stemming from both an interest in game development and a desire to apply Scala's functional programming capabilities in a real-world scenario. Rock-Paper-Scissors serves as an ideal canvas to explore various programming concepts such as event-driven architecture, user interface design, and game logic implementation. Moreover, this project provides an opportunity to delve into animation and transition effects within ScalaFX, enhancing the visual appeal and user experience. By choosing a familiar game, we can focus on optimizing the user interaction and ensuring the software performs efficiently, rather than getting bogged down by overly complex rules or mechanics.

2.0 Presentation Link

https://youtu.be/ghY1yelA-Q4

3.0 UML Diagram



4.0 Functionality

1. Card Selection

- **Player Card Selection**
    - **Description**: Allows the player to select a card from their hand. The selection process involves presenting the player with a visual representation of their available cards and enabling them to choose one through user interaction.
    - **Example Function**: selectCard()
        - **Details**: This function is invoked when the player interacts with a card on the UI. It updates the internal state to reflect the selected card and triggers the card comparison process.

- **Computer Card Selection**
    - **Description**: The computer randomly selects a card from its hand. The selection process is automated and does not require user input.
    - **Example Function**: selectRandomCard()
        - **Details**: This function simulates a random choice by the computer from its available cards. It uses a random number generator to select a card and update the game state accordingly.

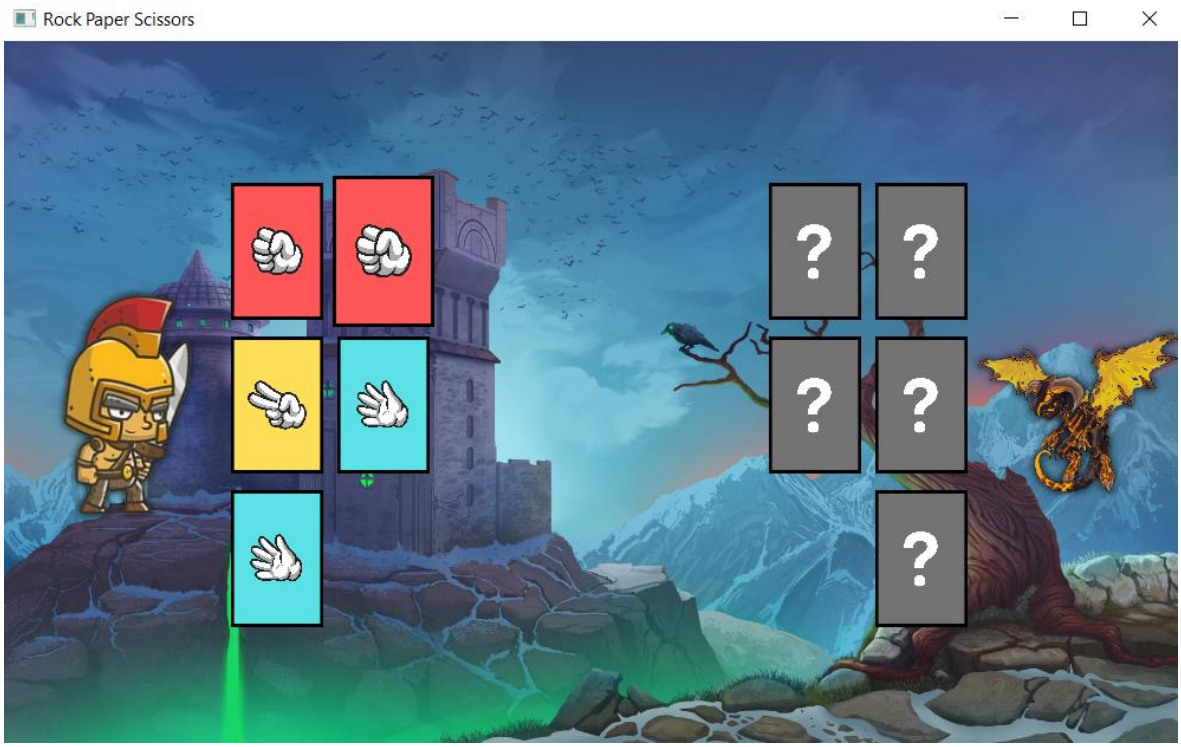


Figure 1: Card Selection

2. Card Comparison

- **Determine Round Outcome**
    - **Description**: Compares the selected cards of the player and the computer to determine the winner of the round based on the established game rules (Rock crushes Scissors, Scissors cuts Paper, Paper covers Rock).
    - **Example Function**: determineWinner()
        - **Details**: This function evaluates the selected cards against the game rules to determine the result of the round. It updates the game state with the outcome (win, lose, or draw) and prepares for the next steps in the game.

3. Card Management

- **Update Decks**
    - **Description**: Updates the player's and computer's card decks after each round. The update involves adding or removing cards based on the round outcome. If the player wins, they gain the computer's card; if the computer wins, the player's card is removed; if it's a draw, both cards are removed.
    - **Example Function**: playRound()
        - **Details**: This function processes the result of the round by modifying both the player's and computer's decks. It handles the logic for card acquisition and removal, reflecting the outcome of the round in the game state.

4. Game Status and Notifications

- **Round Result Popup**
    - **Description**: Displays a popup after each round to indicate the winner and show the cards selected by both the player and the computer. This provides immediate feedback on the result of the round.
    - **Example Function**: showResult()

- **Details**: This function generates a popup window with information about the round's outcome, including which card was selected by whom and the result of the round. It provides a summary of the round for the player.

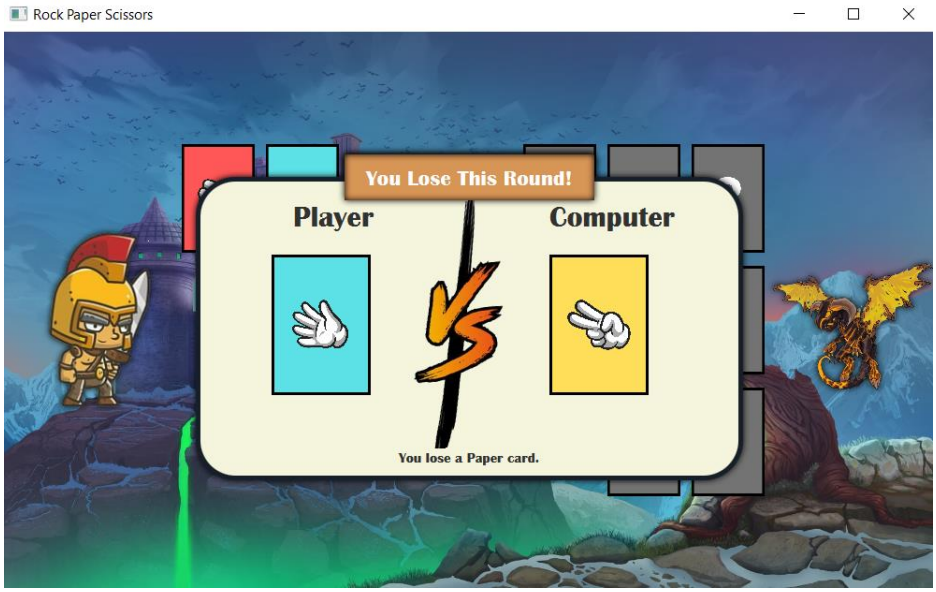


Figure 2: Each Round Result

- **End Game Notification**
  - **Description**: Notifies the player when the game ends, indicating the overall winner when one side runs out of cards. The notification includes one of three possible outcomes: 'You Win,' 'You Lose,' or 'It's a Draw’.
  - **Example Function**: endGame()
    - **Details**: This function triggers a final notification to inform the player of the game’s conclusion. It summarizes the overall result and provides a final message based on the remaining cards.

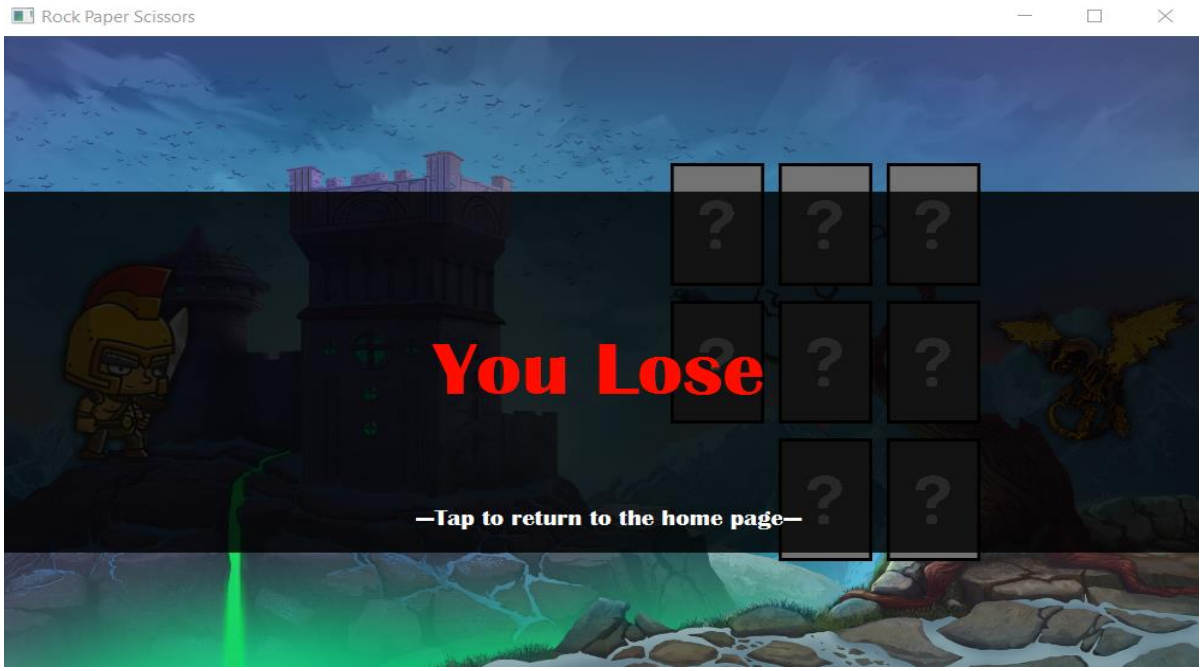


Figure 3: Player Lose Page

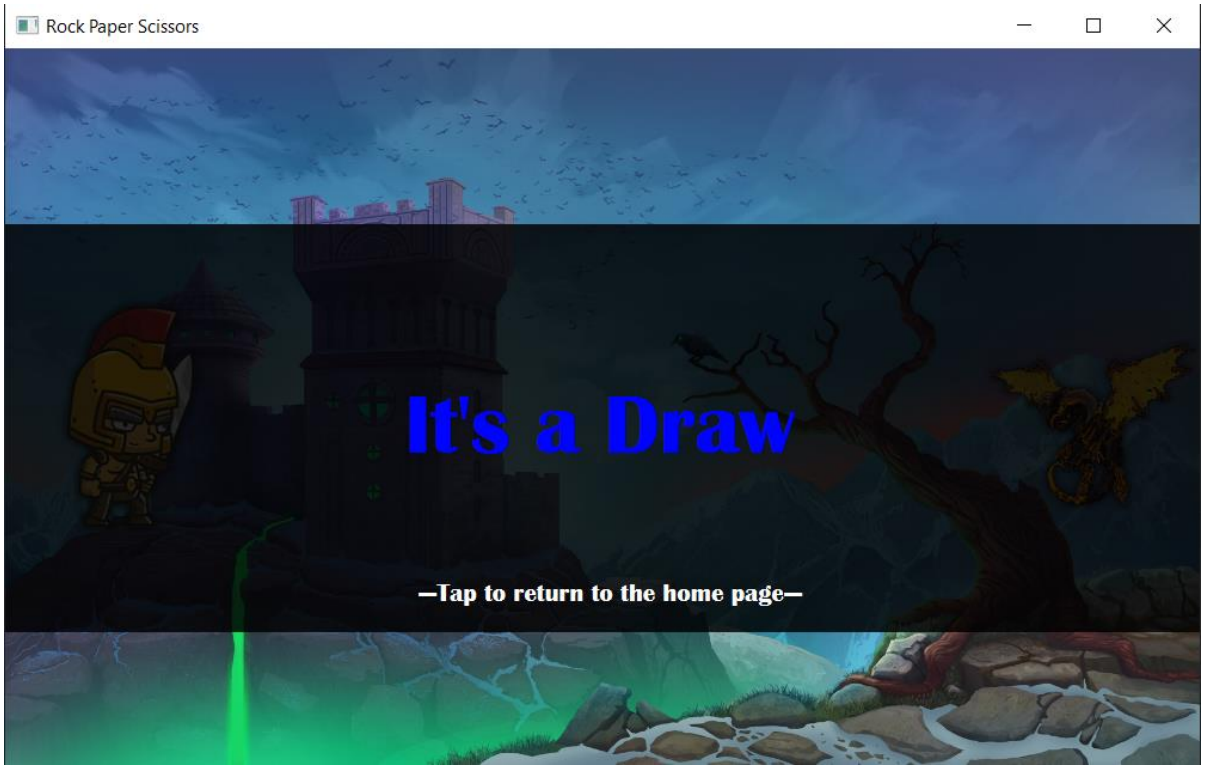


Figure 4: Player and Computer Draw Page

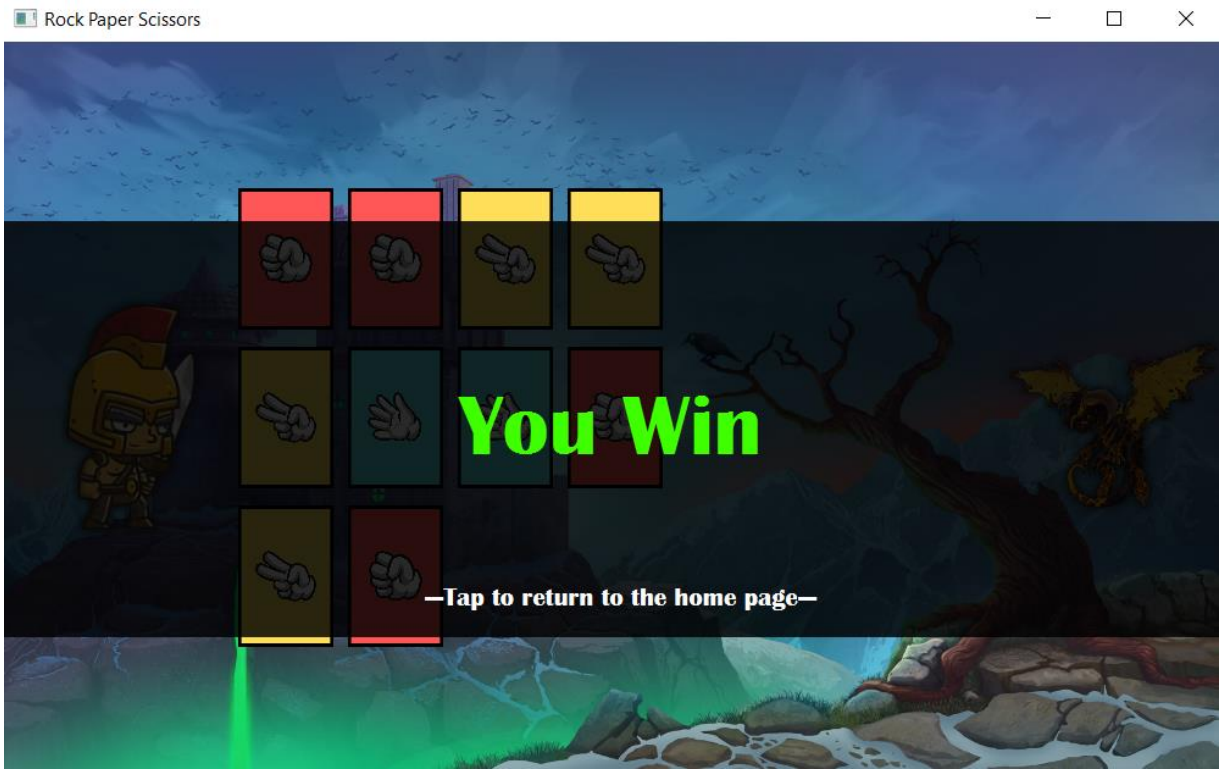


Figure 5: Player Win Page

5. Animation and User Interaction

- **Hover and Click Animations**
  - **Description**: Provides visual feedback for card selection and button interactions through animations. When the user hovers over a card, the card slightly enlarges to indicate it is clickable. Upon clicking, the card performs a quick shrinking animation to signal the selection.
  - **Details**: Animations enhance user interaction by making the interface more responsive and engaging. Hover effects give visual cues, while click animations confirm actions.
  - **Example Function**: addMouseEffects()

- **Transition Screens**
  - **Description**: Displays a transition scene when the game starts, instructing the user to select a card. The transition scene appears for 0.5 seconds before disappearing, providing a smooth visual transition into the game.

- **Details**: This feature introduces the game by showing a temporary scene with instructions or prompts. It ensures the transition between the home page and the game scene is smooth and visually appealing.
- **Example Function**: closeTransitionScene()

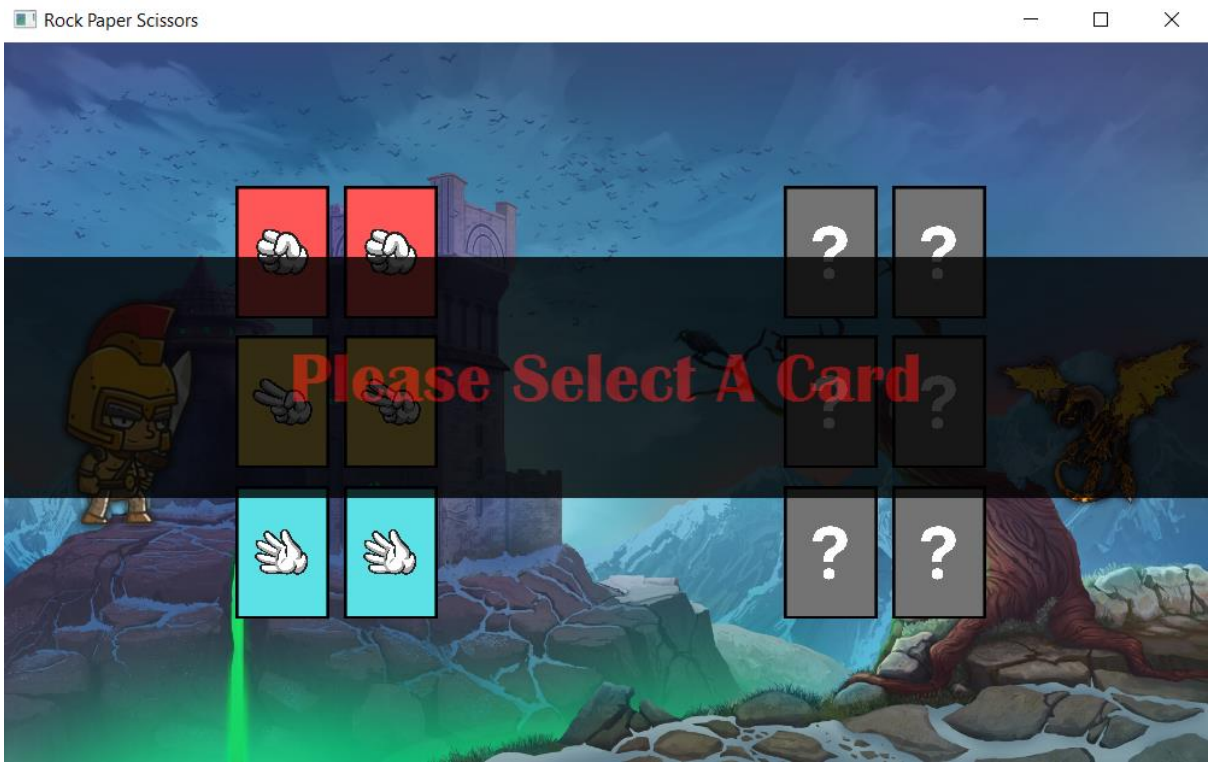


Figure 6: Instruction Pane When Page Start

6. Home Page Management

- **Start Game**
  - **Description**: Initiates gameplay by transitioning from the home page to the game scene. It sets up the game environment and prepares the user for play.
  - **Details**: This feature handles the transition between the home page and the game scene, initializing game variables and preparing the UI for gameplay.
  - **Example Function**: showGameScene()

- **View How To Play**
  - **Description**: Directs users to a section with detailed instructions and rules for the game. It helps users understand the game mechanics and strategies.

- **Details**: This option provides users with access to comprehensive game instructions and tips, ensuring they are well-informed before starting the game.
- **Example Function**: showHowToPlay()

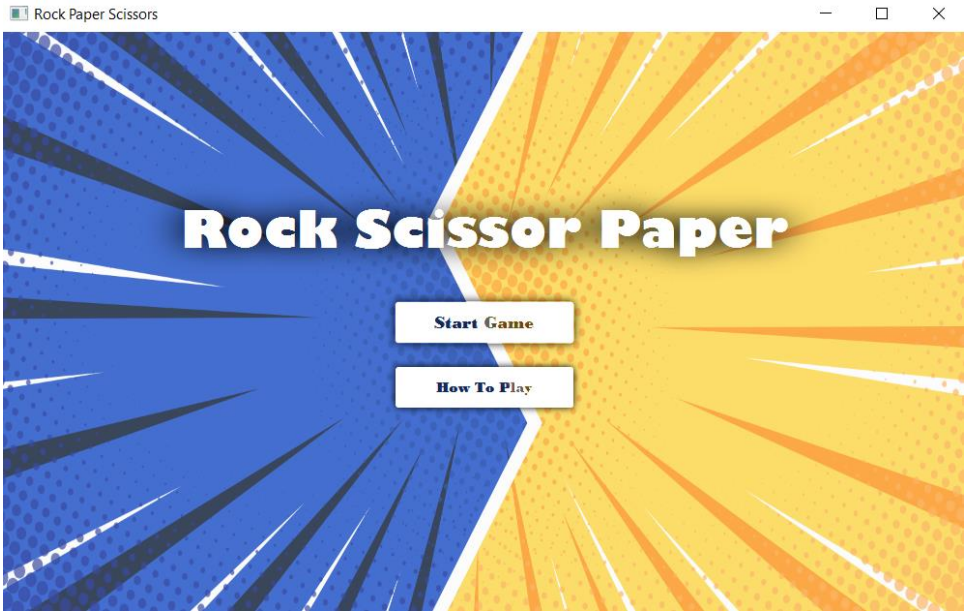


Figure 7: Main Menu Page

Figure 8: How To Play Page

7. UI Management

- **Update Player & Computer Hand UI**
  - **Description**: Clears the existing card pane and creates a new layout for player and computer cards. The layout adjusts dynamically based on the number of cards.
  - **Details**: This function manages the UI presentation of the card hands, recalculating and arranging the cards in the most effective way to fit within the available space.
- **Card Layout**
  - **Description**: Determines the arrangement of cards based on the total number. Cards are displayed in rows and columns, adjusted dynamically to fit the screen.
  - **Details**: The layout logic includes:
    - For up to 6 cards, 2 cards per row.
    - For 7 to 9 cards, 3 cards per row.
    - For more than 9 cards, up to 4 cards per row.
  - **Example Function**: updateHandUI()

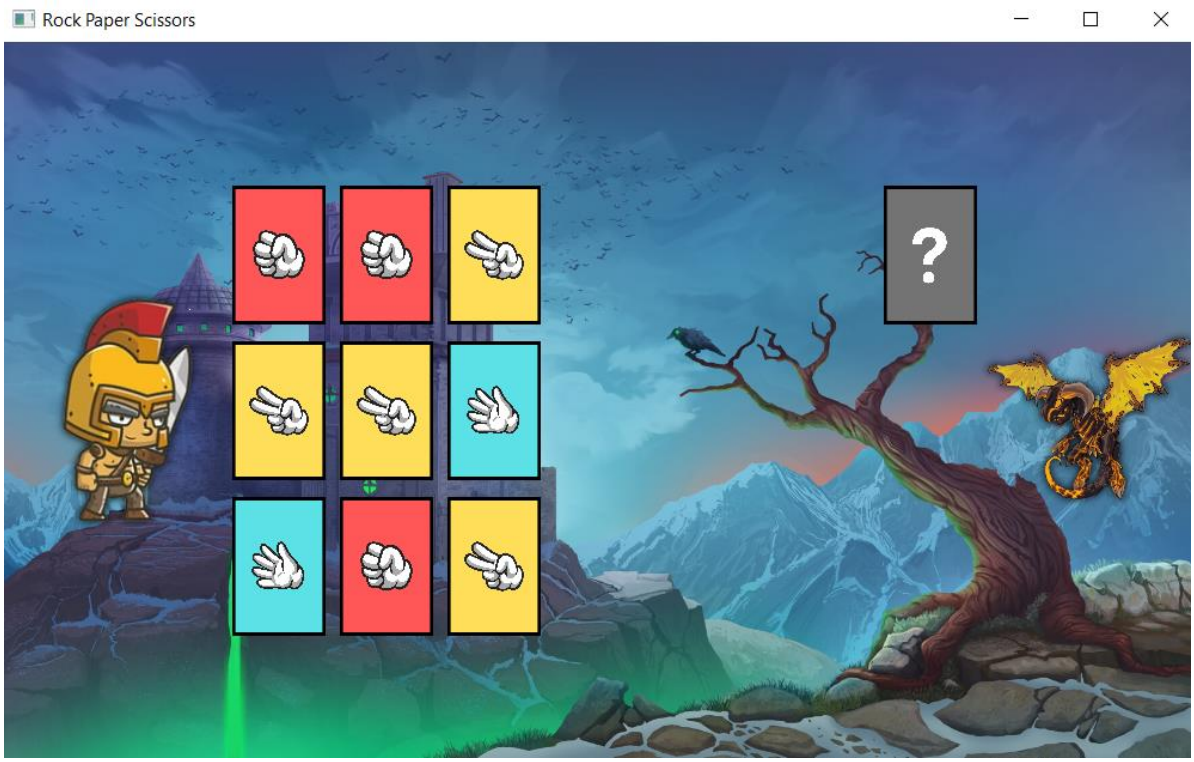


Figure 9: Different Arrangement of Cards

8. Audio Management

- **Background Music**
    - **Description**: Manages the playback of background music across different scenes, ensuring a consistent auditory experience. It starts and loops music for the home page and game scene and stops music when switching scenes.
    - **Example Function**: startBackgroundMusic()
        - **Details**: This function initiates background music playback, looping it as needed and ensuring no overlap between different scenes.
- **Sound Effect**
    - **Description**: Plays short sound effects in response to user interactions, such as button clicks and card selections. This enhances user engagement with auditory feedback.
    - **Example Function**: playSoundEffect()
        - **Details**: This function triggers specific sound effects corresponding to various user actions, providing immediate feedback and enhancing the interactive experience.

5.0 Project File Overview

Folder	File	Function
View	HomePage.fxml	Defines the layout and user interface components for the home page of the application, where users can start the game or view instructions.
	GamePage.fxml	Specifies the layout and user interface components for the game page, including the card images, result display, and game controls.
	HowToPlay.fxml	Outlines the layout and user interface for the "How to Play" page, providing instructions and rules for the game.
Controllers	HomePageController.scala	Manages interactions on the home page, including starting the game and showing the "How To Play" page.
	GamePageController.scala	Manages interactions during the game, including card selection, displaying results, handling game transitions, and ending the game based on the final results.
	HowToPlayController.scala	Handles navigation from the "How to Play" screen back to the home page.
Entities	Card.scala	Defines the card trait and its implementations (Rock, Scissor, Paper) with methods for determining if one card beats or draws with another.
	Player.scala	Represents a player with a collection of cards and methods for selecting, losing, and winning cards.
	ComputerPlayer.scala	Extends the Player class to add functionality for randomly selecting a card for the computer player.
	AudioPlayer.scala	Manages playing sound effects and background music, including starting, stopping, and handling audio resources.

Model	CardUIHelper.scala	Provides utility methods for updating the user interface with card images and applying animations for user interactions.
	GameLogic.scala	Contains the logic for determining the winner of a round and handles the game state updates based on the outcome.
Main.scala		Sets up the primary stage of the ScalaFX application and manages the display of different scenes (HomePage, GamePage, HowToPlay) with background music.

6.0 OOP Concept

1. Encapsulation

Encapsulation ensures that an object's internal state is hidden from the outside world and that only specific methods are exposed to interact with that state. This makes the code more modular and easier to maintain.

Player Class:

- **Internal State Management:** The Player class manages its own cards using a private ListBuffer. External classes interact with this state through public methods like selectCard, loseCard, and winCard. This ensures that the card management logic is self-contained and the internal list is not directly accessible or modifiable from outside the class.

```
class Player {
  private val currentCards: ListBuffer[Card] = ListBuffer.fill(2)(Rock) ++
ListBuffer.fill(2)(Scissor) ++ ListBuffer.fill(2)(Paper)

  def selectCard(card: Card): Option[Card] = {
    if (currentCards.contains(card)) Some(card) else None
  }

  def loseCard(card: Card): Unit = {
    currentCards -= card
  }

  def winCard(card: Card): Unit = {
    currentCards += card
    arrangeCards()
  }

  private def arrangeCards(): Unit = {
    currentCards.sortBy(_.toString).reverse
  }
```

```
  def getCardCount: Int = currentCards.length

  def getCardNames: Seq[String] = currentCards.map(_.toString)
}
```

AudioPlayer Class:

- **Abstraction of Audio Handling:** The AudioPlayer object encapsulates the details of playing sounds and managing audio states. The methods provided (e.g., playSoundEffect, startBackgroundMusic) allow other parts of the application to interact with audio without knowing the underlying implementation details.

```
object AudioPlayer {
  private var activeBackgroundMusic: Option[Clip] = None

  def playSoundEffect(fileUrl: URL): Unit = {
    // Implementation details hidden
  }

  def startBackgroundMusic(fileUrl: URL): Unit = {
    // Implementation details hidden
  }

  def stopCurrentBackgroundMusic(): Unit = {
    // Implementation details hidden
  }
}
```

2. Inheritance

Inheritance allows you to create a new class based on an existing class, inheriting its properties and methods. This promotes code reuse and establishes a natural hierarchy.

Player and ComputerPlayer Classes:

- **Extending Functionality:** The ComputerPlayer class inherits from the Player class. It gains all functionalities of Player, such as card management, while adding its own behavior for selecting a random card.

```
class ComputerPlayer extends Player {
  def selectRandomCard: Card = {
    val randomIndex = Random.nextInt(currentCards.length)
    currentCards(randomIndex)
  }
}
```

3. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexible method calls and interactions.

Card Trait and Subclasses:

- **Flexible Card Interaction:** The Card trait provides a common interface for different card types (Rock, Scissor, Paper). Each card type implements the beats and drawWith methods in its unique way, allowing for flexible interactions between different card types. This polymorphic behavior enables the game logic to interact with any card type using the same method calls, relying on the specific card type's implementation to determine the outcome

```
sealed trait Card {
  def beats(other: Card): Boolean
  def drawWith(other: Card): Boolean
}

case object Rock extends Card {
  override def beats(other: Card): Boolean = other == Scissor
  override def drawWith(other: Card): Boolean = other == Rock
}

case object Scissor extends Card {
  override def beats(other: Card): Boolean = other == Paper
  override def drawWith(other: Card): Boolean = other == Scissor
```

```
}

case object Paper extends Card {
  override def beats(other: Card): Boolean = other == Rock
  override def drawWith(other: Card): Boolean = other == Paper
}
```

4. Abstraction

Abstraction involves defining abstract classes or traits that provide a general template for other classes. It hides the detailed implementation and only exposes the essential features.

Card Trait:

- **General Interface:** The Card trait defines a general template for all card types. The specific details of how each card type behaves (Rock, Scissor, Paper) are abstracted away, allowing the rest of the code to interact with cards through this common interface.

```
sealed trait Card {
  def beats(other: Card): Boolean
  def drawWith(other: Card): Boolean
}
```

7.0 Problems Encountered and Solutions

During the assignment, I encountered several challenges:

1. **FXML Controller Binding:**
    - **Problem:** The FXML elements were not correctly bound to the ScalaFX controller classes, causing the application to fail during runtime. This was primarily due to mismatches between the FXML file and the ScalaFX controller class annotations.
    - **Solution:** To fix this, I ensured that every UI component in the FXML file had a corresponding variable in the ScalaFX controller class, properly annotated with @FXML. I also double-checked that the FXML IDs and variable names were consistent, ensuring proper binding and resolving the runtime errors.
2. **Game Logic Synchronization:**
    - **Problem:** The game's logic was not synchronizing correctly between the player's and the computer's actions. This led to issues where the computer would select its card before the player's action was fully processed, creating an inconsistent game state.
    - **Solution:** I refactored the handleCardPress method to sequence the game actions properly. The player's actions were fully processed and the UI updated before the computer's move was triggered. This ensured the game logic flowed correctly, maintaining a consistent and fair experience for the player.
3. **Animation and UI Feedback:**
    - **Problem:** Implementing smooth and responsive animations in the UI was challenging. Some transitions were not executing in the expected order, which resulted in a suboptimal user experience.
    - **Solution:** I adjusted the timing and sequencing of PauseTransition and FadeTransition animations, ensuring that they executed smoothly and in the intended order. This resulted in a more fluid and visually appealing user interface, enhancing the overall player experience.
4. **Audio Integration and Synchronization:**
    - **Problem:** Integrating sound effects and background music while ensuring they played at the correct moments was another challenge. Some sound effects were either delayed or didn't play at all due to timing issues.

- **Solution:** I encapsulated the audio management within the AudioPlayer class, refining the timing mechanisms within the game logic to trigger sound effects at the appropriate moments. This ensured that audio feedback was synchronized with game events, providing a more immersive experience.

5. **User Input Handling:**
   - **Problem:** Handling simultaneous or rapid user inputs led to unexpected behavior, such as multiple cards being selected at once or animations not completing properly.
   - **Solution:** I implemented input debouncing within the event handlers to manage the timing of user inputs. This prevented multiple actions from being triggered simultaneously, ensuring that the game responded correctly to user interactions.
6. **Game State Management:**
   - **Problem:** Managing the game state, particularly when transitioning between different stages of the game (e.g., from player selection to determining the winner), was complex and prone to errors.
   - **Solution:** I introduced a state management mechanism within the GameLogic class, defining clear stages and transitions for the game. This structure made it easier to track and manage the game state, reducing the likelihood of errors during gameplay.

8.0 Evaluation of Strengths and Weaknesses

Strengths:

1. **Modularity:**
    - The project is well-structured with distinct modules for each part of the game. The use of classes and methods to separate concerns—such as game logic, UI management, and audio handling—ensures that the code is easier to understand, maintain, and extend. This modularity facilitates future enhancements and reduces the risk of introducing bugs when making changes.
2. **User Experience:**
    - The inclusion of responsive animations, well-timed sound effects, and a user-friendly interface enhances the overall experience of the game. The game feels polished and engaging, with visual and auditory feedback providing an immersive environment for the player.
3. **Effective Use of OOP Concepts:**
    - The project demonstrates a strong grasp of object-oriented programming principles. The clear application of encapsulation, inheritance, polymorphism, and composition results in a well-organized codebase that is both flexible and maintainable. The design allows for easy modifications and the potential for future expansion, such as adding new card types or game modes.
4. **Code Reusability:**
    - The project effectively reuses code through inheritance and composition. For example, the ComputerPlayer class extends the Player class, inheriting its properties and methods while adding specific behavior for computer-controlled actions. This reuse of code reduces redundancy and enhances maintainability.
5. **Encapsulation of Complex Behaviors:**
    - Complex behaviors, such as audio management and game state transitions, are encapsulated within dedicated classes like AudioPlayer and GameLogic. This encapsulation hides the complexity from other parts of the program, simplifying the overall design and making the code easier to work with.
6. **Consistent Game Logic:**
    - The game logic is consistent and well-managed, with clear rules for determining the outcome of each round. The use of the Card trait and its

implementations (Rock, Paper, Scissor) provides a flexible and extendable framework for the game's mechanics.

Weaknesses:

1. **Scalability:**
    - The current design is tailored to a simple rock-paper-scissors game. While it works well for this scope, the architecture may not scale efficiently if more complex game mechanics, additional players, or different game modes are introduced. Refactoring may be necessary to accommodate such expansions.
2. **Basic Error Handling:**
    - The error handling throughout the project is minimal. For instance, if an unexpected input occurs or a resource file (like an audio clip) is missing, the game may fail without providing useful feedback to the user. Improving the error handling with more robust checks and user-friendly messages would enhance the game's reliability.
3. **Lack of Automated Testing:**
    - The project does not include automated unit tests, which is a significant limitation. Automated testing would help ensure that each component of the game functions as expected and would make it easier to identify and fix bugs early in the development process. Adding a comprehensive test suite would increase the confidence in the code's correctness and stability.
4. **UI Responsiveness:**
    - While the UI includes animations and visual feedback, there are still some areas where responsiveness could be improved. For example, handling edge cases where the player interacts with the UI in unexpected ways could be refined to prevent any potential glitches or unresponsive behavior.
5. **Limited Flexibility in Game Rules:**
    - The game's rules are hard-coded, which limits flexibility. If there were a need to change the game rules (e.g., introducing new types of cards or altering the winning conditions), significant code changes would be required. Implementing a more dynamic rule system would allow for easier modifications and extensions.

9.0 Conclusion

In conclusion, this object-oriented programming (OOP) project has significantly enhanced my understanding of key principles such as encapsulation, inheritance, and modularity. By structuring the game into distinct classes for game logic, player interactions, and UI components, I've learned the importance of separating concerns to create a more organized and maintainable codebase. Working with ScalaFX also deepened my knowledge of UI design, emphasizing the role of event-driven interactions and animations in delivering a responsive user experience.

Looking ahead, I plan to build on this experience by exploring advanced OOP concepts and design patterns to further improve the scalability and flexibility of my projects. Additionally, I am eager to experiment with more complex game mechanics and AI-driven features, which will challenge me to apply OOP principles to even more intricate and dynamic systems. The skills and insights gained from this project will undoubtedly serve as a strong foundation for my future endeavors in software development.

10.0 References

Anthropic. (2023). Claude (Oct 8 version) [Large language model]. https://www.anthropic.com/

AnuraDsgn. (2019, January 13). *Wooden photo frame template - cartoon style*. iStock. https://www.istockphoto.com/vector/wooden-photo-frame-template-cartoon-style-gm1093587582-293479667

Cole, M. (n.d.). *Enchanted forest landscape background*. Vecteezy. https://www.vecteezy.com/vector-art/3821981-enchanted-forest-landscape-background

Concept art Video Games Drawing Character Illustration, 2d character transparent background PNG clipart | HiClipart. (n.d.). https://www.hiclipart.com/free-transparent-background-png-clipart-ffpdh

Der Test. (2023, April 24). *Victory Sound Effect* [Video]. YouTube. https://www.youtube.com/watch?v=teUWsONJkk8

Free Gaming Sound Effects. (2021b, November 7). *Select Button Sound Effect* [Video]. YouTube. https://www.youtube.com/watch?v=KsDg-ggEOvk

Gaming Sound FX. (2015, December 2). *The Price is Right Losing Horn - Sound Effect (HD)* [Video]. YouTube. https://www.youtube.com/watch?v=_asNhzXq72w

Gaming Sound FX. (2016, September 10). *Strategy (Dramatic Music) - Gaming Background Music (HD)* [Video]. YouTube. https://www.youtube.com/watch?v=UpqywJ48Mi4

Home, A. (2023, September 13). *Right Arrow free icons designed by Andrean Prabowo*. Pinterest. https://in.pinterest.com/pin/866591153285035485/

Houbor. (2021, March 9). *Bright Yellow Background Beautiful Blue White Cartoon Anime Vs*. Pikbest. https://pikbest.com/templates/bright-yellow-background-beautiful-blue-white-cartoon-anime-vs_6605782.html

Jordankzf. (n.d.). *blackjack-scala-gui/src/main/scala/blackjack/view at main · jordankzf/blackjack-scala-gui*. GitHub. https://github.com/jordankzf/blackjack-scala-gui/tree/main/src/main/scala/blackjack/view

Mike Sovin. (n.d.). *Backgrounds for a card game*. ArtStation. https://mikesovin.artstation.com/projects/gJQaZE

MusiMan. (2024, July 11). *Demon Slayer | Infinity Castle Theme (But The Best Part Is Looped)* [Video]. YouTube. https://www.youtube.com/watch?v=ZW6D6mm-mKc

OpenAI. (2023). *ChatGPT* (Mar 14 version) [Large language model]. https://chat.openai.com/chat

Premium Vector | Versus or vs letters image. (2022, March 31). Freepik. https://www.freepik.com/premium-vector/versus-vs-letters-image_25222468.htm

Rladstaetter. (n.d.). *GitHub - rladstaetter/fx-tictactoe: A simple tic tac toe game written in scala and javafx*. GitHub. https://github.com/rladstaetter/fx-tictactoe

Rock Paper Scissors – Game Instructions | JAX Games. (n.d.). https://www.jaxgames.com/rock-paper-scissors-game-instructions/

Sound Effect Database. (2020, October 12). *Menu Game Button Click Sound Effect* [Video]. YouTube. https://www.youtube.com/watch?v=yxafINGGm4Y

Sound Library. (2021, March 5). *Gamemode Select - Sound Effect for editing* [Video]. YouTube. https://www.youtube.com/watch?v=Vq1qeApsZCA

Wiki, C. T. M. (n.d.). *Lord Magma*. Monsterwars Wiki.https://monsterwars.fandom.com/wiki/Lord_Magma