s3804620

Tran Huy Khanh

COSC2659 - iOS Development
Assignment 2

Table of Contents:

I. Introduction:

With many themes that everyone can use to build their game application. For games like board game or dice game, we also need to use calculations to be able to win the game. So, why can't I just develop a game about math? And from there, my ideas and motivation began to develop because of that thought.

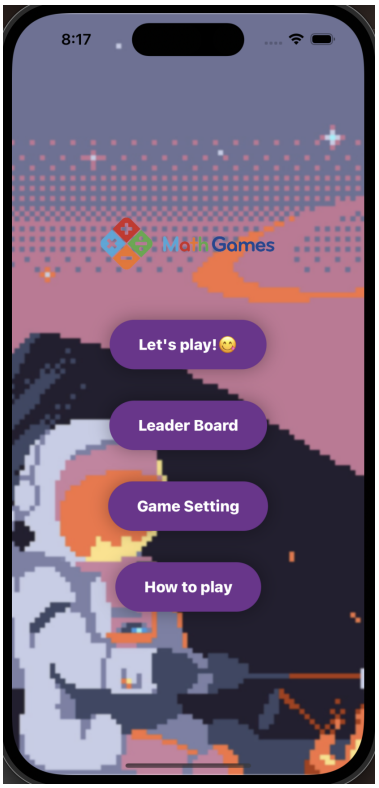
II. Project Description:

In this math game, it's all about doing the right calculation and choosing the right answer. At first glance, the game feels very easy, but I have built some functions to make the game feel more difficult. This game is divided into 3 different game modes. Therefore, to win the game, the player needs to achieve the score required by the game to win.

III. Implementation Details:

1. Main Features:

a. Menu view:



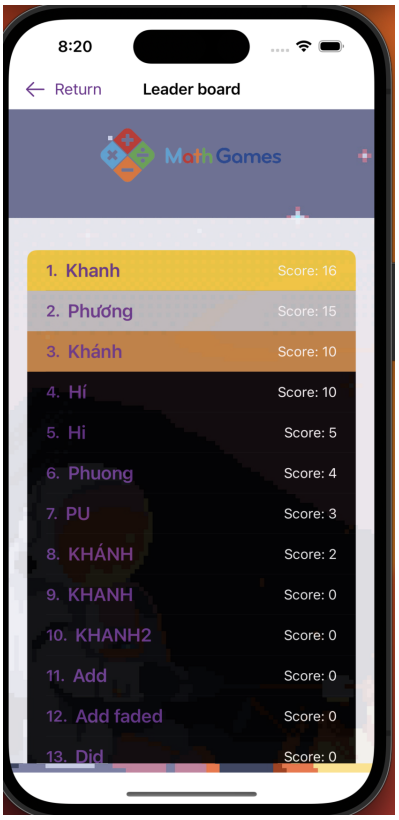
MenuView is designed using ZStack and VStack layouts. In addition, to create links with other pages, I dropped the designs into NavigationView{} and added NavigationLink{} in the design between PrimaryButton() to be able to access the pages I want to visit. Besides that, I have used properties to make this app work better like:

- '@StateObject private var LeaderboardRefresh' so that it can help in managing leaderboards, it follows the ObservableObject protocol, which means it can publish changes to its properties, making them available accessible to other views.

- @Binding var userName: allows the parent view to pass data to this view and receive updates from that view.
- @AppStorage properties: These properties are used to store user settings like the selected game mode, dark mode preference, and game language. @AppStorage is a property wrapper that automatically saves and retrieves values from the app's UserDefaults.

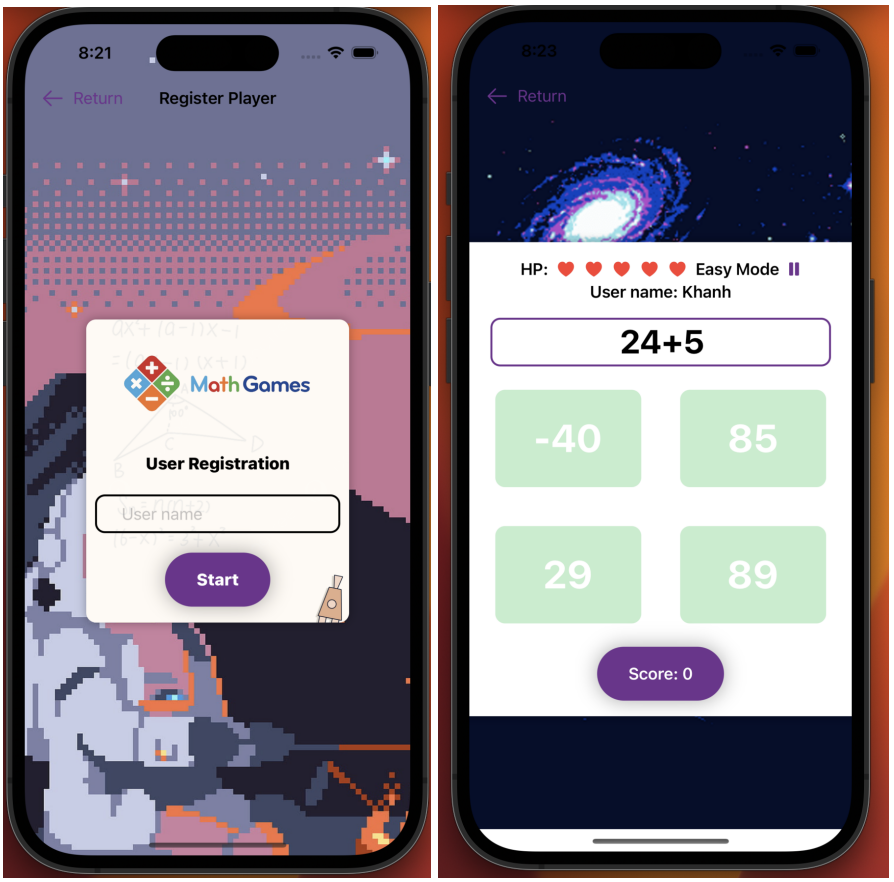
And finally use .onApear to trigger playSound to mount the music file.

b. Leaderboard view:



On this Leaderboard view, using '@EnvironmentObject private var leaderboardRefresh' to create an environment for the access object to refresh the leaderboard. Leaderboard fetches score data via the ScoreManager.shared.getLeaderboard() call. This function can retrieve scores and usernames from several data sources and then populate the leaderboard. Users can view their rankings, usernames, scores and related achievements in an attractive format. View supports both light and dark modes, plays background music, and provides an intuitive interface for users to navigate leaderboards and explore their achievements. Tapping a user entry brings up an achievement popup with additional details. Overall, it enhances the gaming experience by celebrating the player's achievements and competitiveness.

c. Game view:



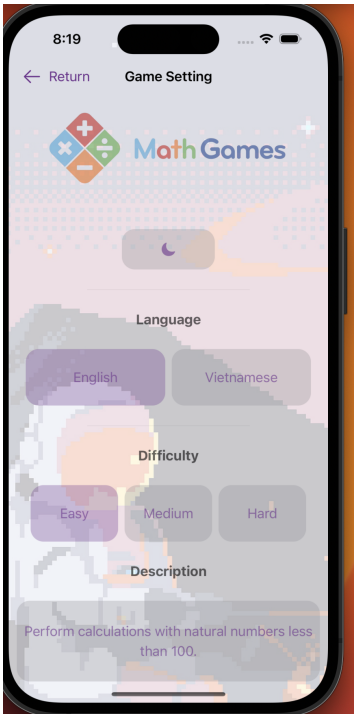
Before going to GameView, we need to register username. So the RegisterView will appear first, the main functionality of this view revolves around assisting users in entering their chosen username. The user enters their desired username via the TextField and this information is captured and stored in the userName state variable. Furthermore, RegisterView is designed to support multiple languages, with user interface elements and text messages adaptable to both English and Vietnamese, as defined by the gameLanguage variable. The navigation function of the view depends on the selected game mode (gameMode). Based on this selection, the user is redirected to a specific destination view (destinationView). Available target views include GameView, Game1View and Game2View, Each view corresponds to a different difficulty level of the game. If none of these conditions match then it looks like there is an incomplete section of code.

In Game View, using @State to manage the dynamic behavior of the view. So this page contains a lot of components to build the game. The game presents a mathematical equation to the user with randomly generated numbers and calculations (addition, subtraction, multiplication, division).

Users choose from multiple response options presented as buttons. If the user chooses the correct answer, their score will increase.

If the user chooses an incorrect answer, their health will decrease, a health reduction animation will occur. The game tracks the user's progress and checks win or lose conditions. Here the game is divided into 3 different modes and is divided into different win conditions. In easy mode, it is 10 points, medium mode is 15, and finally the hardest is 20.

d. Game Setting View:

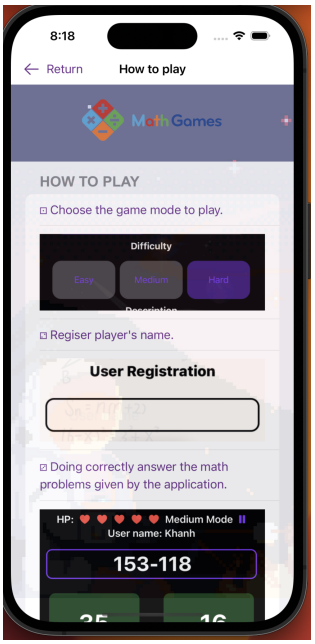


GameSettingView is a SwiftUI-based view in an iOS app responsible for configuring game settings. Here is a comprehensive explanation of the code. This view allows users to customize their game experience with a variety of settings:

- Dark mode: Users can turn dark mode on/off, adjusting the app's appearance according to their preferences. Using @AppStorage("isDarkMode") to be able to perform functions and create Buttons for adjusting dark/light mod.
- Language: Users can choose the game language between English and Vietnamese, helping the application reach a wider audience. In building language builds for options, I use @Binding var gameLanguage and @AppStorage to manage operations. I then made a conditional based on the language selection then the text in the body could change according to the @Biding I called.
- Difficulty Level: Users can choose from three difficulty levels: Easy, Medium and Hard, each level has a different complexity in mathematical challenges.

View presents these options in a visually appealing interface with buttons and descriptions. Changes to the settings will update @AppStorage values accordingly, ensuring that user preferences persist between app launches.

e. How To Play View:



HowToPlayView in SwiftUI provides a concise and user-friendly guide on how to play iOS games. It supports both English and Vietnamese, ensuring accessibility to a wider audience. Users can choose their preferred game mode, register their player name, solve math problems and aim for high scores. The view also provides essential information about the app such as name, course, year of publication, and location of development. With attractive visuals and intuitive navigation, this view enhances the overall user experience.

2. Advance Features:

a. Save and Resume:

First I create a Model file, then create a Swift vs file named GameOperationAndState.swift. In this file, I create a struct GameState used to store sroce and currentHealth. In GameView, I have 2 functions are savegame() and loadgame(), which are responsible for saving and loading the state of the game so that players can pause and resume their gameplay. These functions utilize UserDefaults and JSONEncoder/JSONDecoder to achieve this functionality.

In savegame(), this function is used to save the current state of the game when the player decides to pause or exit the game. It creates an instance of a custom structure called GameState, which represents the state of the game. This structure can contain properties such as score, currentHealth, and possibly other properties related to the state of the game. Next, it tries to encode this GameState structure into JSON data format using JSONEncode. JSON encoding is a way to convert Swift objects (like GameState) into a format that can be saved or transferred. If encryption is successful, the resulting JSON data will be saved to the user's device using UserDefaults. It is stored under a specific key, "savedGameState", allowing the data to be retrieved later.

In loadgame(), this function is used to load the previously saved game state when the player wants to continue his game. It tries to retrieve data from UserDefaults using the "savedGameState" key. If data is found and successfully decoded using JSONDecoding, it is converted back to a GameState object. The game's current score and Health are then updated based on the loaded state, allowing players to continue where they left off. The isGameOver is usually set to false because when the game is saved and loaded, the player is assumed to intend to continue playing.

In body view, I display 2 buttons:

- “Play” button: displayed when a saved game state exists. Tapping it loads the saved game state, allowing the player to pick up where they left off.
- “Pause” button: Displayed when no saved game states exist (usually when starting a new game). Tapping it saves the current game state, allowing the player to continue the game later.

b. Game Progression and Levels:

```
if score < 5 { // set the game level higher when score < 5
        operation = [.addition, .subtraction].randomElement() ?? .addition
    } else {
        operation = MathOperation.random()
    }
```

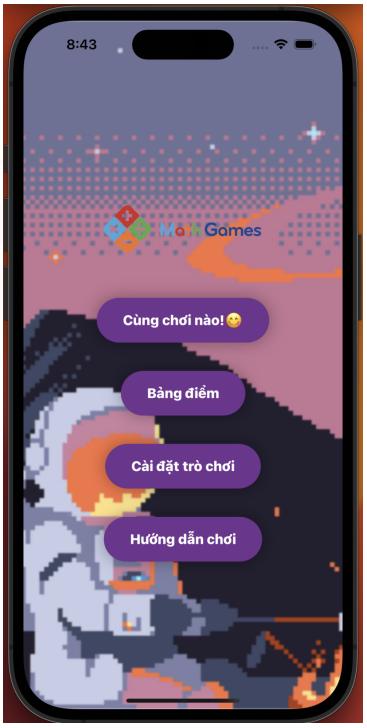
```
if score < 3 {
        operation = [.addition, .subtraction].randomElement() ?? .addition
} else if score < 6 {
    operation = [.addition, .subtraction, .multiplication].randomElement() ?? .addition
} else {
        operation = MathOperation.random()
    }
```

```
if score < 2 {
    operation = [.addition, .subtraction].randomElement() ?? .addition
} else if score < 4 {
    operation = [.addition, .subtraction, .multiplication].randomElement() ?? .addition
} else {
        operation = MathOperation.random()
    }

if score < 3 {
    operation1 = [.addition, .subtraction].randomElement() ?? .addition
} else if score < 5 {
    operation1 = [.addition, .subtraction, .multiplication].randomElement() ?? .addition
} else {
        operation1 = MathOperation.random()
    }
```

In the Game Progression and Levels establishment, in each game mode. I use the scores the player achieves to gradually increase the level. In easy mode, I consider the condition that when the score is below 5, there are only 2 calculations, addition and subtraction. If the score is higher, all the established calculations will occur. In medium mode, I consider scores below 3, only addition and subtraction, below 6 points, add multiplication and higher scores, similar to easy mode. And finally in hard mode, here I create 2 operations to use for 2 calculations in turn and set it as in the other 2 modes

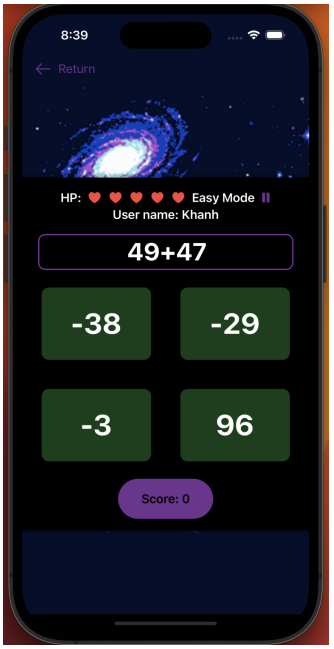
c. Multiple Language Support:



@Binding var gameLanguage: String: A binding property for the selected game language. Then I use @AppStorage("englishIsClicked") var englishIsClicked: Bool = true and @AppStorage("vietnameseIsClicked") var vietnameseIsClicked: Bool = false. By using this I

have set up if else condition when player click on language option all languages will be converted in text.

d. Toggle Theme Setting:



‘@AppStorage("isDarkMode") private var isDarkMode = false’ : This property uses the @AppStorage property wrapper to store and retrieve the isDarkMode boolean value in UserDefaults. And finally, the environment palette is set based on the isDark state using the ".environment" modifier.

IV. Conclusion:

The core of a math game is the game logic, which includes creating problems, evaluating user input, and providing feedback. Swift's powerful and flexible standard library makes it suitable for implementing various game mechanics and algorithms. In short, building a math game in Swift offers a great opportunity to combine coding skills with educational goals. With the right approach, you can create a fun and valuable learning tool for players of all ages.

V. Reference:

Tom Huynh, SSET-Contact-List-iOS, https://github.com/TomHuynhSG/RMIT-Casino-iOS , accessed Sept,2023

Video demo:
https://drive.google.com/file/d/1G3yXJE6vU_kMBL8QtwTuYyrMuK5135Pj/view?usp=sharing