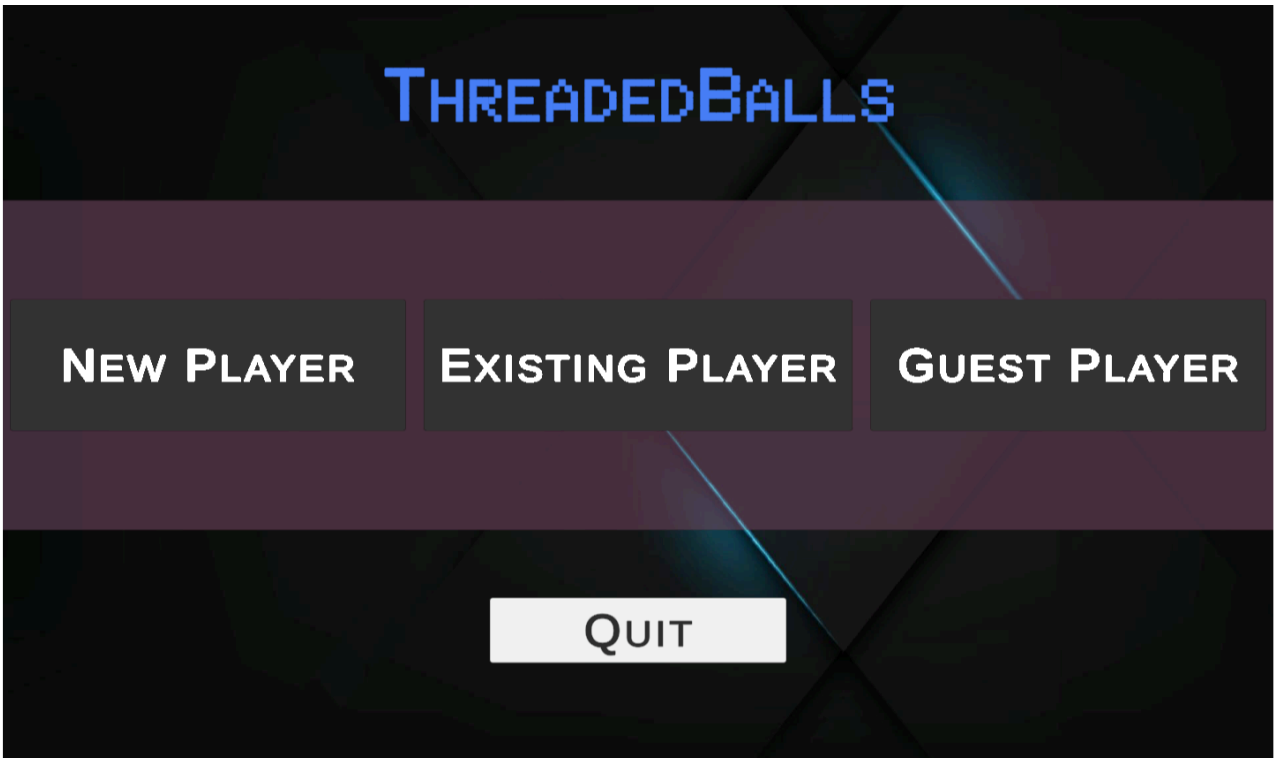
Integrating Concurrency, Databases, and FSM Operations in a Responsive 2D Unity Game

Game Name: ThreadedBalls ; **Developer:** Dennis Yiaile ; HOW TOPLAY

Github Link: https://github.com/Dennis-tec/ThreadedBallUnityGame

Azure Portal Link(To just Display The Interface):
https://threadedballs-ata9h7aaa3hpbpgu.eastus-01.azurewebsites.net/



The game has been designed exclusively for Windows due to database interactions. It doesn't function on the web because Unity doesn't support direct interactions with databases from web platforms. While you can view the interface on the Azure Portal, the game will not work if you press buttons that require database interactions.

Table of Contents

1 Objective.....3

2 Motivations.....3

3 Goals.....4

4 About The Game.....4

5 Tools & Technologies Purposes.....5

1 Unity.....5

2 Microsoft SQL.....5

3 C#.....5

4 Visual Studio.....5

5 .NET Framework.....6

6 FSM.....6

6 Game Interface/How To Play.....6

6.1 Step 1: Home Menu.....6

6.2 Step 2: Main Menu.....7

6.3 Step 3: Settings Menu.....8

6.4 Step 4: Play Menu.....9

7 FSM CONTEXT.....13

8 DATABASES.....15

9 CONCURRENCY.....16

10 Challenges.....18

11 Key Learnings & Takeaways.....18

1 Objective

- This project will test the integration of Finite State Machines (FSM), concurrency, and database management within a Unity-based game. The aim is to leverage FSM for structured game state management, concurrency to optimize real-time interactions, and databases to store and retrieve players and game data.

- Proposal

2 Motivations

- I desire to enhance my programming/technical skills through hands-on, real-world applications while gaining deeper insights into the design and development processes within the software engineering field. This project will allow me to bridge theoretical knowledge with practical experience, equipping me with the tools and understanding necessary to build software applications in the real world.

- Interest in learning a new programming language, software development framework, IDE, database systems and SQL queries, and game development framework.

3 Goals

- Develop a responsive 2D game that effectively combines Finite State Machines (FSM), concurrency, and database management. This integration will ensure smooth operation and maintain consistent game states by preventing race conditions and avoiding inconsistencies, even under concurrent event interactions.

- Implement a system that accurately tracks and manages player actions, game objects, and game state transitions. This will ensure a responsive experience by synchronizing

game logic with the database and FSM, allowing seamless transitions between game states while preserving data correctness and enhancing gameplay fluidity.

4 About The Game

The player is a rectangular object placed at the bottom of the screen when the game starts. This player can move up, down, left, or right to hit a falling ball to earn points. When the player hits the ball at a higher height, a coin pops up and the player can also hit the ball to earn extra points. These balls and coins are simply running threads. Any ball falling is an instance of a thread running and any coin that pops up is a thread running. Players can pause, restart, resume, or quit an ongoing game OR move to the next level, restart, or quit the current game session after the game ends.

5 Tools & Technologies Purposes

1 Unity

- ★ Cross-platform game development engine to develop the game interface.
- ★ Develop and design responsive 2D game interfaces and objects(i.e balls) using the platform.
- ★ Use to connect the game interfaces/scenes and databases to control game actions.

2 Microsoft SQL

- ★ Use a simple relational SQL database system to manage and store players' and game data.
- ★ Write SQL code to add, update, and get data from the SQL database.

3 C#

- ★ Code that controls the game logic and objects is written in this programming language
- ★ Each game object(i.e player and balls) is controlled by C# scripts/files
- ★ Multithreading capabilities are written in C#

4 Visual Studio

- ★ IDE to be used: Best for large-scale projects, including game development with Unity, better support for debugging, and .NET
- ★ Platform to write, debug, and test C# code and its functionalities

5 .NET Framework

- ★ Combine C# programming language with .NET to build secure, reliable, and cross-platform high-performance game applications

6 FSM

- ★ Apply FSM contexts to handle game state and transitions between game menus such as (from Main Menu to Settings)

6 Game Interface/How To Play

6.1 Step 1: Home Menu

This is the first game menu where a player chooses to create a NEW PLAYER, use EXISTING PLAYERS, QUIT, or PLAY AS GUEST by pressing the buttons shown below. The name that will be selected/created will be recorded in the database and any game completed by the player would be recorded in the database for that player.

Fig 1: Home Menu Scene(First/Start Game Session)

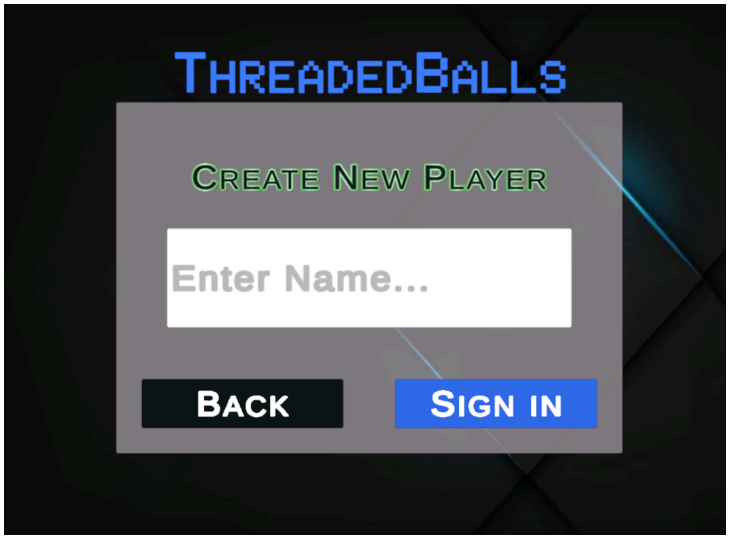


Fig 2: Create a New Player Menu

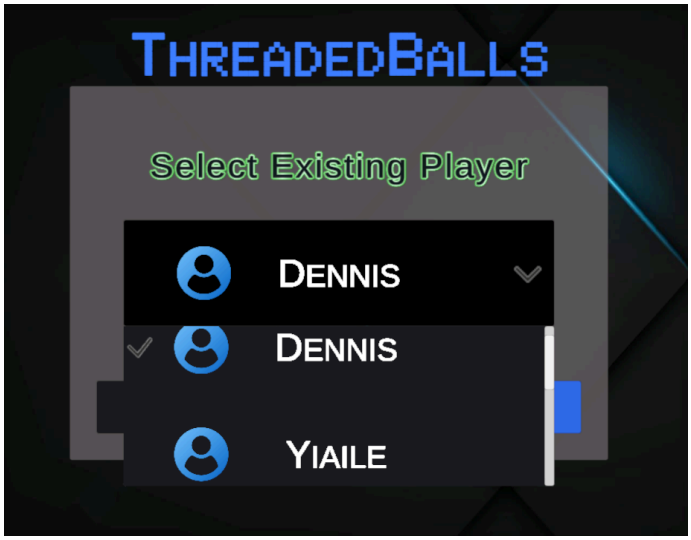


Fig 3: Existing Players Menu

6.2 Step 2: Main Menu

When the player is selected/created and one decides to sign in from the Home Menu, they will be taken to this Main Menu by pressing a button. The name of the selected/created player will

be displayed in the top left corner of this Main Menu. From here, the player can decide to move to the settings menu by pressing the SETTINGS button, go back to the Home menu by pressing the HOME button, or start playing by pressing the PLAY button.

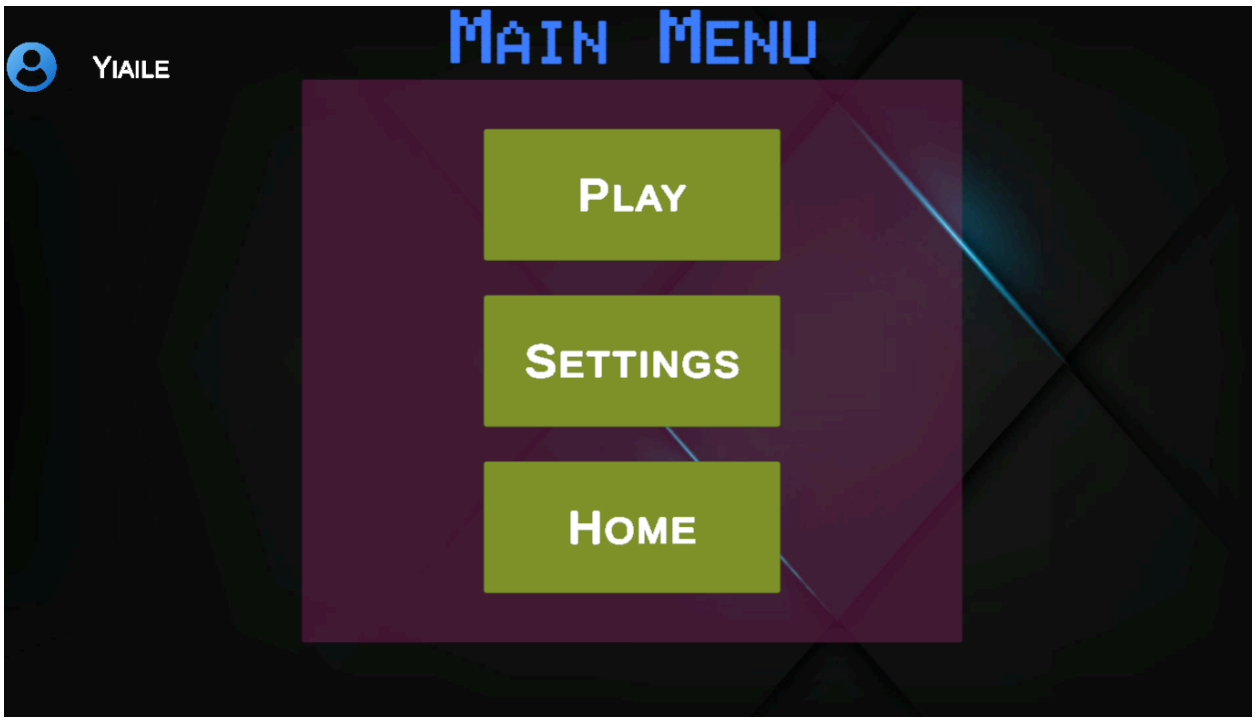


Fig 4: Main Menu Scene

6.3 Step 3: Settings Menu

This is where the players can modify the game by selecting the level where they want to play. After selecting the level, the players can go back to the MAIN MENU by pressing the BACK button and start playing the game There are three different levels as explained below:

Levels: Games get difficult based on the different levels below.

- Easy**:** The balls fall at lower frequencies although the frequency increases as the game progresses.
- Medium: The balls fall at higher frequencies than the ones in the easy level but lower than the high level. Also, the frequency increases as the game progresses

- High: The balls fall at higher frequencies than any level and the frequency increases as the game progresses.

Fig 5: Settings Menu Scene

6.4 Step 4: Play Menu

During the play, the text of the level will be displayed on the screen, and the game will display a three-second timer before the game starts officially. Below are the scores that will also be displayed on the game scene:

- **Scores**:
  - Current Scores: While playing, the player sees their scores on the top right corner of the screen as they increase based on the number of balls hit.
  - High Scores: This shows the highest number of points garnered by any player in the current game level. This is retrieved from the database

- New High Scores: This shows the new high scores after the current player attained higher than the current high scores(replaces the high score).

Fig 6: Game Starting Timer Scene

The following steps occur when the game starts:

- Balls Fall from the top bar/top of the interface and the player ensures that they hit them to earn points, no points earned if they fail to hit them. The higher the number of balls hit, the higher the scores.
- Balls fall randomly from any point at the top of the bar and the player must be attentive to hit every ball by running at the place where the ball will fall.
- A ball that is not hit will disappear at the bottom and there will be no points to earn.
- The frequency of ball falls will increase as the game progresses.
- If the player hits a ball at a higher height, the coin pops and they earn extra points if they hit the coins.

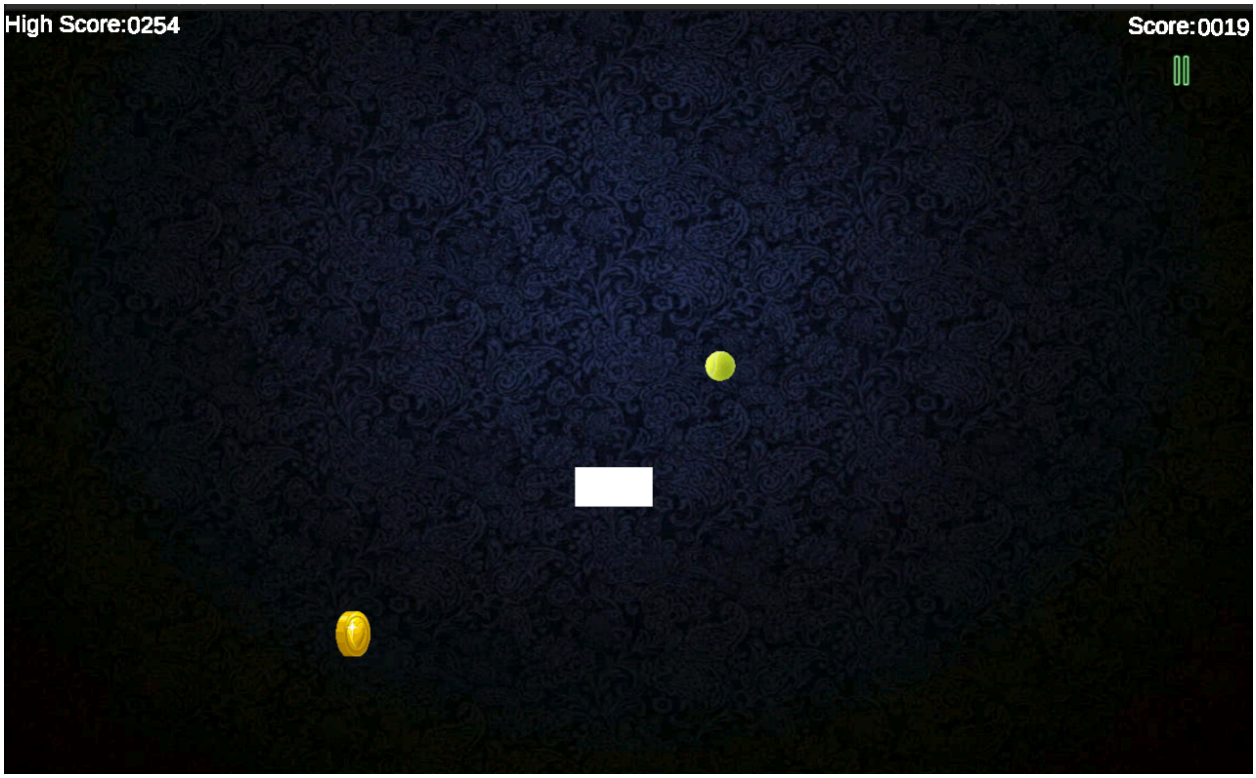


Fig 7: Game Player In Session Scene

Players can pause the game by pressing the || (pause) button. They will be directed to the pause menu and from there, they can restart the game by pressing the RESTART button, quit the game by pressing the MENU button, or RESUME the current game.

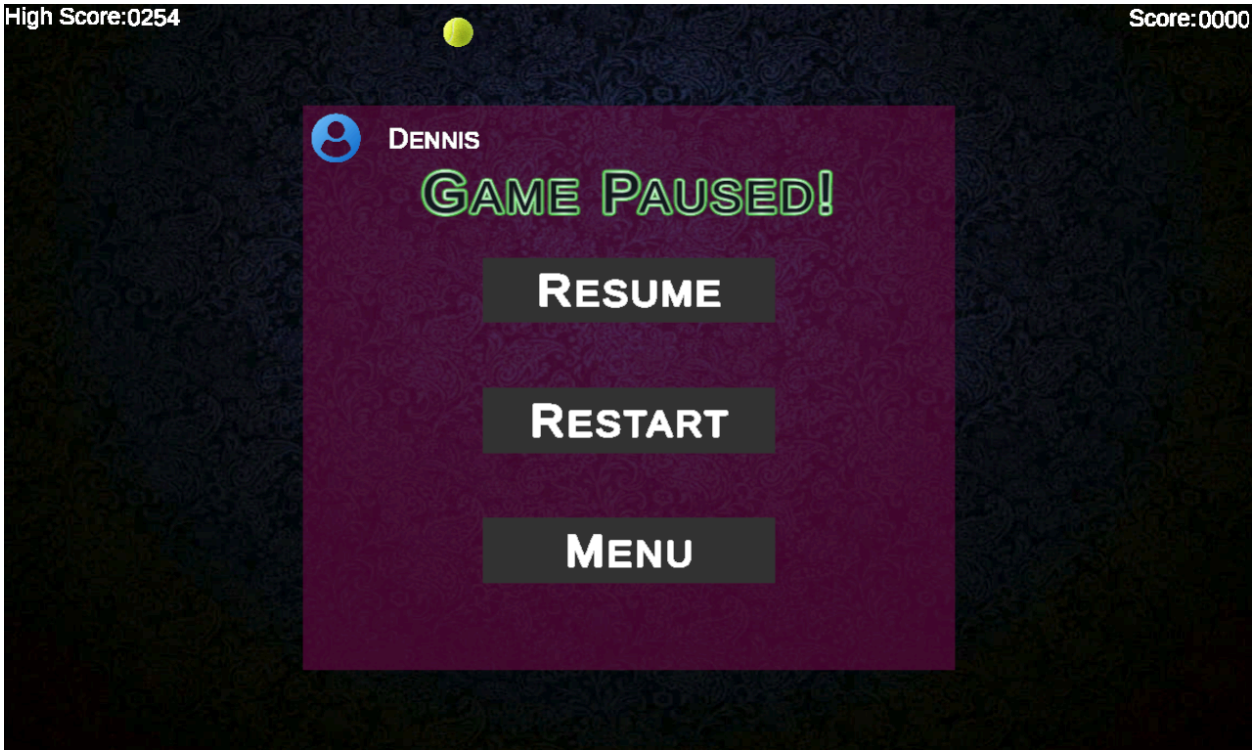


Fig 8: Game Paused Menu Scene

After the game ends, the player is directed to the Game Completed Menu, where they can move to the next level by pressing the Next LEVEL button, restart the game by pressing the RESTART button, or quit the game by pressing the Main Menu button. The previous game score, player ID, level name, and game ID will be recorded in the database when the game is completed.

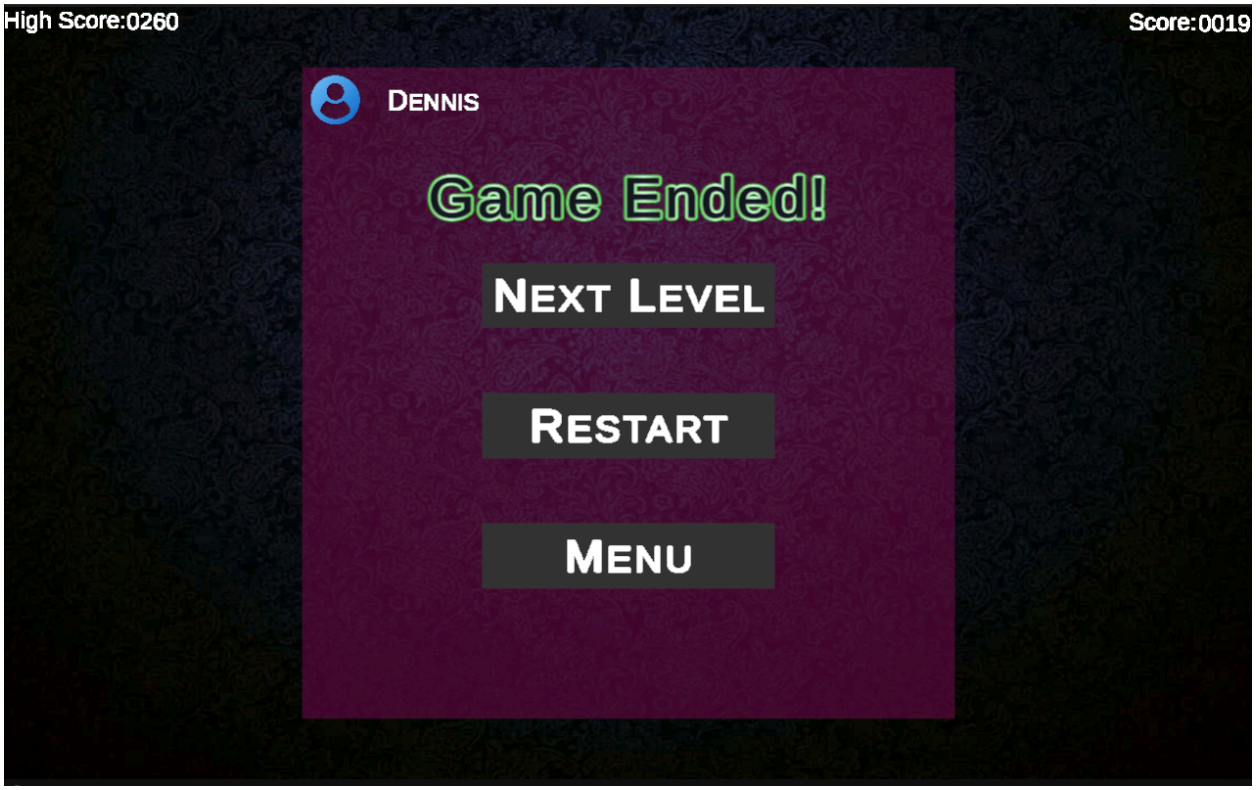


Fig 9: Game Completed Menu

7 FSM CONTEXT

I used UnityEngine.SceneManagement module to help transition between game states by pressing buttons that move the current game session to another. Below is a screenshot of a C# script/file that manages this system. When a button is pressed, a function that matches the pressed button within this file(Scripts/StateTransitions.cs) is called to initiate and execute a transition. For instance, to load to the settings menu, the SETTINGS button is pressed on the MAIN MENU. In the back-end, this calls the MoveToSettings() function in the Scripts/StateTransitions.cs C# file to load the Settings Menu

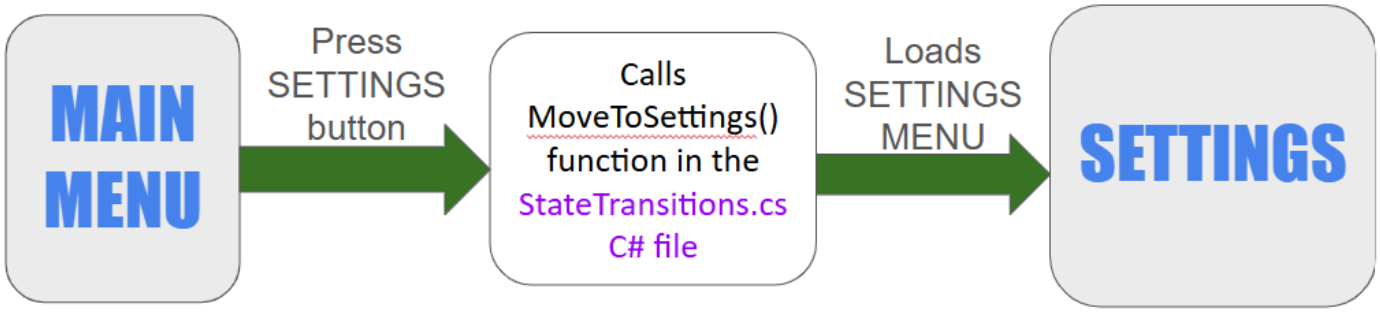


Fig 10: Shows what happens in the back end when a SETTING button is pressed on the MAIN MENU to load the SETTINGS menu

```
using UnityEngine.SceneManagement;

Unity Script (4 asset references) | 3 references
public class StateTransitions : MonoBehaviour
{

    2 references
    public void PlayButton()
    {
        SceneManager.LoadScene("Game Play");
    }

    0 references
    public void MoveToSettings()
    {
        SceneManager.LoadScene("Settings");
    }

    0 references
    public void ReturnHome()
    {
        SceneManager.LoadScene("Home");
    }
    0 references
    public void QuitGame()
    {
        UnityEditor.EditorApplication.isPlaying = false;
        Application.Quit();
        Debug.Log("Game is exiting");
    }

    3 references
    public static void GoToMainMenu()
    {
        SceneManager.LoadScene("Main Menu");
        PauseGame.GamePaused = false;
        BallsGenerator.GameCompleted = false;
    }
```

Fig 11: Functions that control Game Transitions based on the press of a button.

8 DATABASES

Integrated Microsoft SQL Server Database to manage the players' and game data. I did this by creating two relational database tables: Players and GamesRecord table. The players' table stores the player ID and the players' name. When a new player is created, they will be recorded in this table within the database. The second table is the GamesRecord table which stores information about each game played. This includes the game ID, level name, player ID(NULL if the player is a Guest), and the score for that game. When a game is completed, it's recorded in this table. A couple of other operations that rely on the database include getting the highest score from the database for each level, getting a list of existing players, and checking if the new name being created is within the database. These operations are handled by functions within the Scripts/SqlDatabase.cs C# file. For instance, to get the maximum score from the database, the function InteractWithDatabase(int operation) is first called by supplying the GetHighestScoreOperation operation which then calls the GetHighScoresFromTheDatabase(SqlConnection connection, string gamesTable, string levelName) function that runs a SQL query to retrieve the highest score from the Games Table in the database as shown in the screenshots below.

```
public void InteractWithDatabase(int operation)
{
    string connectionString = $"Server={ServerName};Database={DatabaseName};User ID=Dyiaile;Password=Yiaile6345!;TrustServerCertificate=True;";
    try
    {
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            connection.Open();
            Debug.Log("Successfully Connected to the Database!");
            if (operation == SqlDatabaseConstants.CreateTablesOperation)
            {
                CreateTables(connection, playersTable, gamesRecordTable);
            }
            else if (operation == SqlDatabaseConstants.RegisterNewPlayerOperation)
            {
                RegisterNewPlayerInTheDatabase(connection, playersTable, UserInputs.NewUserInput);
            }
            else if (operation == SqlDatabaseConstants.RecordCompletedGameOperation)
            {
                RecordCompletedGameInTheDatabase(connection, gamesRecordTable, BallConstants.GetNameForLevel(), SqlDatabaseConstants.CompletedGameScore);
            }
            else if (operation == SqlDatabaseConstants.GetRegisteredPlayersOperation)
            {
                GetRegisteredPlayers(connection, playersTable);
            }
            else if (operation == SqlDatabaseConstants.GetHighestScoreOperation)
            {
                GetHighScoresFromTheDatabase(connection, gamesRecordTable, BallConstants.GetNameForLevel());
            }
            else if (operation == SqlDatabaseConstants.GetUserNameIdOperations)
            {
                GetUserNameId(connection, playersTable, SqlDatabaseConstants.ActiveUserName);
            }
        }
    }
    catch (Exception ex)
    {
```

Fig 12: Function that controls database interactions based on the operation's parameters

```
1 reference
public void GetHighScoresFromTheDatabase(SqlConnection connection, string gamesTable, string levelName)
{
    string maxScoreQuery = $@"SELECT ISNULL(MAX(Scores), 0) AS MaxScore FROM {gamesTable} where Level_Name = '{levelName}'";

    using (SqlCommand command = new SqlCommand(maxScoreQuery, connection))
    {
        object result = command.ExecuteScalar();
        SqlDatabaseConstants.RecordedHighScore = Convert.ToInt32(result);
    }
}
```

Fig 13: Screenshot of the function that retrieves the maximum score from the table that contains details about games already played per level

9 CONCURRENCY

As mentioned earlier, this is a threaded ball game, where threads control the number of balls falling per game session. In total, we have 50 threads that run anytime a game starts which means that we will have 50 balls falling per game. However, this can lead to a situation where multiple threads are running at the same time causing multiple balls to fall at the same time and the player will not be able to hit all these balls. To prevent this from happening, I used a

semaphore and sleep timer to ensure mutual exclusion. Each thread will have to acquire a thread before instantiating a ball to fall. This semaphore can only acquired by one thread at a time and other threads can only acquire it after the current thread releases the semaphore. This means that only one thread can instantiate a ball at a time so only one ball can fall at a time. Before releasing a semaphore, the thread will sleep for some time to ensure that the player gets more time to hit the current ball before the next ball falls. However, this sleep time reduces as the game progresses to have balls falling more frequently after another making the game harder. This game logic is handled within the Scripts/BallGenerator.cs C# file/script as shown in the screenshot below.

```
        // Delay the start of ball creation
        Invoke("StartBallCreationWithDelay", 3.3f);
    }
    0 references
    private void StartBallCreationWithDelay()
    {
        // Start the thread for ball creation
        ExecuteBallThreadsOnUnityMainThread.Instance();
        Thread ballThread = new Thread(() => CreateBallOnMainThread(6f, "Thread 1"));
        Thread ballThread2 = new Thread(() => CreateBallOnMainThread(6f, "Thread 2"));
        Thread ballThread3 = new Thread(() => CreateBallOnMainThread(6f, "Thread 3"));
        Thread ballThread4 = new Thread(() => CreateBallOnMainThread(6f, "Thread 4"));
        Thread ballThread5 = new Thread(() => CreateBallOnMainThread(6f, "Thread 5"));
        ballThread.Start();
        ballThread2.Start();
        ballThread3.Start();
        ballThread4.Start();
        ballThread5.Start();
    }
```

Fig 14: Number of Threads Running/Starting.

```
5 references
private void CreateBallOnMainThread(float y, string threadId)
{
    // add the function to a list of function to be executed in the main thread
    for (global::System.Int32 i = 0; i < 10; i++)
    {
        try
        {
            semaphore.WaitOne();
            Debug.Log(threadId + " Acquire semaphore");
            pauseThread.WaitOne();  // Pause the thread if the game is paused

            lock (criticalSectionLock)
            {
                Debug.Log(threadId + " Entering critical section");
                ExecuteBallThreadsOnUnityMainThread.Instance().Enqueue(() => InstantiateBall(y, threadId));
                Debug.Log(threadId + " Leaving critical section");
            }
        }
        finally
        {
            pauseThread.WaitOne(); // Pause the threads if the game is paused

            Thread.Sleep(threadsSleepTime);
            threadsSleepTime -= timeDecreament;
            semaphore.Release();
            Debug.Log(threadId + " Releases semaphore");
        }
    }
}
```

Fig 15: Critical Section where only one Thread Acquire and Release a Semaphore at a time.

10 Challenges

- It was difficult to ensure mutual exclusion - Ensure only one thread runs at a time and instantiate a ball to fall. But using semaphores helped.

11 Key Learnings & Takeaways

- C#: This is a new programming language that I learned to use
- Unity: I learned to use and explore this new game development platform and use it to create game scenes, and game objects, and ensure smooth transitions between game sessions.

- I learned to connect Microsoft SQL databases with unity to ensure that the players and game data are correctly recorded between game sessions.
- I learned to use SQL query language and Visual Studio IDE