PROJECT REPORT
ON
SOCCER SHOOTING GAME

(FOR THE COURSE OF T. Y. B.SC. COMPUTER SCIENCE)

DESIGNED AND DEVELOPED
BY
Mr. VIHANG RAJENDRA PIMPALE

GUIDED BY
Ms. PRATIKSHA HARWARKAR
DEPARTMENT OF COMPUTER SCIENCE

PARLE TILAK VIDYALAYA ASSOCIATION'S
MULUND COLLEGE OF COMMERCE (AUTONOMOUS)
(AFFILIATED TO UNIVERSITY OF MUMBAI)
2024-2025

MULUND (WEST), MUMBAI-400080
MAHARASHTRA, INDIA

ACKNOWLEDGEMENT

I have a great pleasure in representing this project report entitled “Soccer Shooting Game” and I grab this opportunity to convey my immense regards towards all the distinguished people who have their valuable contribution in the hour of need.

I like to extend my gratitude to our beloved Principal Dr. Sonali Pednekar for her timely and prestigious guidance.

I take this opportunity to thank Dr. Reena Nagda, Coordinator of the Department and all the faculty members of the Department of Computer Science of Mulund College of Commerce, for giving me an opportunity to complete this project and the most needed guidance throughout the duration of the programme.

I am extremely grateful to my project guide Ms. Pratiksha Harwalkar for her valuable guidance and necessary support during each phase of the project. She was the source of continuous encouragement as each milestone was crossed.

Finally, I also owe to my fellow friends who have been a constant source of help to solve the problems that cropped up during the project development process.

VIHANG RAJENDRA PIMPALE

PLAGARISM CHECKER:

Plagiarism Checker X Originality Report

Plagiarism Checker X - Report

Originality Assessment

2%

Overall Similarity

Date: Mar 9, 2025
Matches: 400 / 16146 words
Sources: 3

Remarks: Low similarity detected, check with your supervisor if changes are required.

Verify Report:
View Certificate Online

Plagiarism Checker X

Certificate of Plagiarism

This is to certify that scanned report is valid and has

2% Similarity.

v 8.0.7 Words Matched : 400 / Words Scanned : 16146

***Dated* : March 09, 2025.**

This certificate has been produced by Plagiarism Checker X.
plagiarismcheckerx.com | plagx.com

INDEX

1.Title 1
1.1 Title of Project 1
1.2 Type of Project 1
1.3 Developed by 1
2. Introduction 2
2.1 Objective of Project 2
2.2 Description of the Current System 2
2.3 Limitations of the Current System 2
2.4 Description of the Proposed System 3
2.5 Advantages of the Proposed System 3
3. Requirement Specification 4
3.1 Software Requirements 4
3.2 Hardware Requirements 4
3.4 Fact Finding Questions 5
4.System Design: 7
4.1 Event Table: 7
4.2 Class Diagram: 8
4.3 Usecase Diagram: 10
4.4 Sequence Diagram: 12
4.5 Activity Diagram: 15
4.6. State Diagram 18
4.7 Package diagram: 19
4.8 Component diagram: 20
4.9 Deployment diagram: 22
4.10 Database Design: 23
-players: 23
5.System Implementation: 25
C# Files: 25
5.1.FreekickCore.cs: 25
5.2.BallCollisionHandler.cs: 31
5.3.GameManager.cs: 33
5.4.CameraController.cs: 40
5.5.Login.cs: 42
5.6.Registration.cs: 45
5.7.Ball.cs: 46

5.8.GoalKeeper.cs: 51
5.9.InterceptShotMainState.cs: 63
5.10.PunchBallMainState.cs: 76
5.11.TendGoalMainState: 82
5.12. pauseMenu.cs: 88
5.13.freekicklev.cs: 91
6.Results 94
6.1 Validation and Naming Conventions: 94
6.2 Screenshots: 95
6.2.1 Registration scene: 95
6.2.2 Register page: 97
6.2.3 Login Scene: 97
6.2.4 Start menu scene: 99
6.2.5 levels scene: 101
6.2.6 Freekick practice mode: 102
6.2.7 Penalty-Shootout mode: 104
6.2.8 freekick-levels scene: 108
6.2.9 Level 1: 109
6.2.10 level 2: 110
6.2.11 level 3: 111
6.2.12 level 4: 111
6.2.13 level 5: 112
6.2.14 level 6: 113
6.2.15 level 7: 114
6.2.16 level 8: 115
6.2.17 level 9: 116
7. Future Enhancement & conclusion: 117
7.1 Future Enhancement: 118
7.2 Conclusions: 118
8.References: 120
9 Annexures 121
9.1 Figure list 121

1.Title

1.1 Title of Project
SOCCER-SHOT

1.2 Type of Project
SOCCER SHOOTING GAME

1.3 Developed by
VIHANG PIMPALE

2. Introduction

2.1 Objective of Project

- Realistic Free Kick and Penalty Experience**:** Develop an engaging football game that accurately simulates free kick and penalty shootout scenarios, offering players a realistic and immersive gameplay experience.
- Multiple Game Modes: Implement different gameplay modes, including Unlimited Penalty Mode**,** Free Kick Practice Mode and Free Kick Levels, where players must hit targets to progress through stages.
- Database Login System**:** Integrate a Login system for player to access game not allowing unregistered players to play the game.
- Dynamic Difficulty and AI Goalkeeping**:** Enhance gameplay by introducing dynamic difficulty adjustments and AI-controlled goalkeepers that respond intelligently to player shots, creating a challenging experience.
- Customizable Settings and Controls**:** Allow players to adjust game settings such as ball curve, shot power, and difficulty level, along with customizable controls for a personalized gaming experience.
- Engaging UI and Audio Experience**:** Design a user-friendly interface with a pause menu that includes options like resume, music toggle, and return to levels, complemented by immersive sound effects and background music.

2.2 Description of the Current System

- Football Stadium Environment**:** The current system features a 3D football stadium environment
- Free Kick Shooting System: Players can take free kicks with a customizable ball curve, shot power, and target aiming. In this mode players can move around the stadium and aim and shoot the ball
- User Interface and Menus: The game includes a main menu**,** level selection screen**, and** pause menu **with** options such as resume, toggle music, and return to levels**.**
- Physics-Based Ball Movement: The football mechanics use physics, ensuring that ball movement and curve shots**,** behave naturally.
- Background Music and Sound Effects: The game features sound effects**,** including crowd noises and goal celebrations.
- Game Menu**:** The game features a game menu to navigate throughout the game with a pause button, resume button and a quit button that exits the game and closes the application.

2.3 Limitations of the Current System

- Limited Free Kick Mechanics**:** Current system may not offer precise ball curve control or realistic physics, making their free kick system feel less immersive than yours.
- No Freekick Levels**:** current system lacks proper free-kick, levels leaving players wanting for more content.
- Basic AI Goalkeeping**:** current system has predictable or poorly balanced goalkeepers, either making saves too easy or impossible, whereas your game has **a** more realistic and challenging AI goalie**.**

- No Music Control**:** current system doesn't allow players to toggle background music, reducing overall flexibility and accessibility.
- No Intuitive UI: Current System does not feature an interactive and easy to use user interface.

2.4 Description of the Proposed System

- Free Kick Shooting System: Players can take free kicks with a customizable ball curve, shot power, and target aiming. In free kick levels, players must hit designated targets to complete the stage.
- Penalty Shootout Mode**:** The game features a penalty shootout mode where players attempt to score against an AI-controlled goalkeeper. This mode includes a goals counter.
- User Interface and Menus: The game includes a main menu**,** level selection **screen**, and pause menu with options such as resume, toggle music, and return to levels**.** The UI is designed for easy navigation and a seamless user experience.
- Physics-Based Ball Movement**:** The football mechanics use realistic physics, ensuring that ball movement, curve shots, and goalkeeper reactions behave naturally, enhancing gameplay authenticity.
- Background Music and Sound Effects**:** The game features dynamic sound effects, including ball kicks, crowd noises, and goal celebrations. Players can toggle background music on or off via the pause menu.

2.5 Advantages of the Proposed System

- Realistic Free Kick Mechanics: Current System offers precise ball control**,** including curving, aiming, and power adjustments**,** making free kicks feel more natural and skill-based**.**
- Target-Hitting Challenge Mode: Unlike standard penalty shootouts, proposed System includes unique free kick levels where players must hit targets, adding variety and skill-based progression.
- Better Goalkeeper AI: The goalkeeper reacts dynamically to shots rather than following pre-scripted movements, making saves more realistic and challenging**.**
- Pause Menu with Functional Controls: Proposed system allows players to pause, toggle music, and return to the level menu smoothly, which is often missing in other free kick-based games.
- Smooth and Intuitive Controls**:** The ball movement, shooting mechanics, and UI are optimized for responsiveness**,** ensuring a better gameplay experience.
- Enhanced Immersion with Background Music: Players can enjoy background music, improving the overall atmosphere while giving them the option to toggle it on or off.

3. Requirement Specification

3.1 Software Requirements

- Operating System:
- Windows 10/11, macOS, or Linux (for development)
- Development Tools:
- Unity Game Engine – Cross-platform game framework
- Visual Studio Code – Preferred IDE for game development
- Database & Backend Services:
- MySQL – For real-time data storage and retrieval
- MAMP server-to access and activate database services

3.2 Hardware Requirements

- Minimum System Requirements:
- Processor: Intel Core i3 (7th generation or newer) / AMD Ryzen 3 or equivalent.
- Memory: 4GB RAM minimum, 8GB RAM recommended for smoother operation.
- Storage: HDD for game resources
- Display: 1366 x 768 resolution display or higher.
- Input Devices:
    - Keyboard and Mouse: Standard input devices for navigation and game controls.
- Audio Output: Speakers or headphones for background music.

- Recommended System Requirements:
- Processor: Intel Core i5 (10th generation or newer) / AMD Ryzen 5 or equivalent.
- Memory: 16GB RAM for optimal performance, especially when multitasking.
- Storage: SSD storage for faster application loading and response times.
- Display: Full HD (1920 x 1080) or higher resolution display.
- Input Devices:
    - Keyboard and Mouse: Standard input devices for navigation and
    - game controls.
- Audio Output: Quality speakers or headphones for Background Music

3.3 Data Requirements

- Username
- Password

3.4 Fact Finding Questions

- Gameplay and User Experience Questions:
- What are the core gameplay mechanics that players expect in a football game?
    - Research into popular football games revealed key mechanics such as dribbling, passing, shooting, goalkeeping, and AI-controlled opponents.
    - Finding: Players prefer intuitive and responsive controls that allow for skill-based gameplay.
- What factors contribute to player engagement and long-term retention?
    - Investigation into user preferences showed that skill progression, challenges, unlockable content, and competitive multiplayer modes are essential for retention.
    - Finding: A level-based system with increasing difficulty, leaderboard integration, and unlockable customizations enhances player motivation.
- How can the game ensure realistic and immersive football physics?
    - Analysis of physics engines and ball movement mechanics in existing games.
    - Finding: Implementing realistic ball physics, player animations, and environmental effects (such as wind and pitch conditions) improves immersion.
- What kind of AI should be implemented for goalkeepers and defenders?
    - Research into AI decision-making for football simulations, including positioning, reaction speed, and difficulty scaling.
    - Finding: AI goalkeepers and defenders should adapt based on player skill level to maintain a fair and competitive experience.
- What customization options do players expect in a football game?
    - Surveys on player preferences regarding in-game customization.
    - Finding: Customizable teams, player appearances, and control schemes enhance personalization and engagement.
- Technical Implementation Questions:
- What is the best approach for implementing smooth and responsive player movement?
    - Comparison of movement algorithms such as Rigidbody physics vs. character controller-based movement.
    - Finding: A hybrid approach using Rigidbody for ball interactions and character controllers for player movement provides the best balance of realism and control.

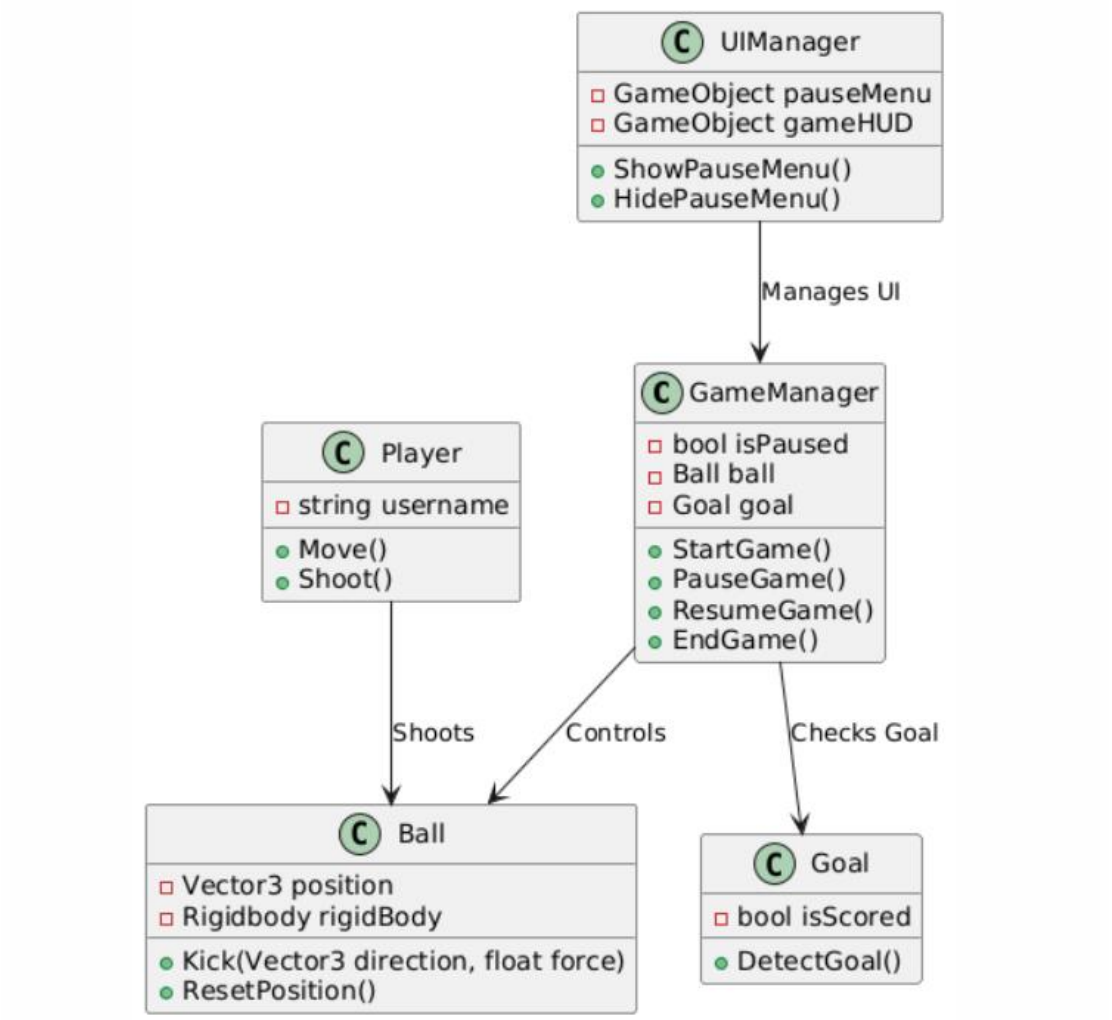
- How can the game ensure consistent performance across different hardware configurations?
    - Research into optimization techniques such as level-of-detail (LOD) rendering and physics scaling.
    - Finding: Using adaptive graphics settings and physics-based performance adjustments ensures smooth gameplay on both low-end and high-end systems.
- What methods should be used to detect and handle ball collisions accurately?
    - Testing different collision detection techniques, including Raycasting and Rigidbody physics.
    - Finding: A combination of physics-based collisions and event-driven triggers provides accurate and responsive ball interactions.
- How can animations be synchronized with gameplay mechanics?
    - Investigation into animation blending and inverse kinematics for realistic player movements.
    - Finding: Using animation state machines combined with physics-based movement ensures fluid and natural transitions.
- Game Mode and Content Integration Questions:
- What game modes should be included to offer variety and replayability?
    - Research into popular football game modes such as penalty shootouts, free kick challenges, and full match simulations.
    - Finding: Implementing penalty shootouts, free kick levels, unlimited play mode, and a time-based challenge mode provides a diverse gameplay experience.
- How can the level structure be designed for progression and difficulty scaling?
    - Evaluation of difficulty scaling methods, such as AI improvement and shot precision requirements.
    - Finding: Gradually increasing goalkeeper difficulty, introducing obstacles, and requiring precise shots create a balanced challenge curve.
- What visual and audio elements are necessary for an immersive football experience?
    - Analysis of sound design and environmental effects in football games.
    - Finding: Realistic crowd sounds, ball impact effects, and stadium ambiance contribute significantly to immersion.
- UI and User Interaction Questions:
- How can the game's user interface (UI) be optimized for accessibility and ease of use?
    - Research into best practices for sports game UI design.
    - Finding: A clean and minimalistic UI with intuitive menu navigation ensures a smooth player experience.
- How can a pause menu be effectively implemented without disrupting immersion?
    - Analysis of pause menu designs in existing football games.
    - Finding: A semi-transparent overlay with quick access to resume, settings, and exit options provides the best balance of accessibility and immersion.

4.System Design:

4.1 Event Table:

Event Name	Trigger Condition	Action Performed	Outcome
Game Start	Player clicks "Start Game" button	Loads the selected game mode	Game scene initializes
Ball Kicked	Player presses the shoot button	Ball moves towards the target	Determines goal/miss
Goal Scored	Ball enters the goal	Updates score, resets ball position	Score increases
Free Kick Target Hit	Ball collides with the target	Displays "Level Completed" panel	Player moves to next level
Pause Button Pressed	Player clicks "Pause" button	Displays pause menu	Game pauses
Resume Game	Player clicks "Resume" button	Hides pause menu, resumes gameplay	Game continues
Go Back to Levels	Player selects "Go Back" in menu	Loads level selection screen	Player returns to menu
Toggle Music	Player Selects “toggle music” in pause menu	Turns off the background music	Music is disabled

4.2 Class Diagram:



GameManager Class

The gamemanager class controls the overall game flow, including pausing and resuming gameplay. It interacts with the ball, goal, and UI to manage the game state.

- Attributes: Manages ball, goal, and score.
- Methods: Includes methods like StartGame(), PauseGame(), and EndGame() to control game flow.

Player Class

The Player class allows user interaction through movement and shooting actions.

- Attributes: Stores the player's username.
- Methods: Move() for player movement and Shoot() for triggering the ball-kicking action.
- Relationship: Interacts directly with the Ball class.

Ball Class

The Ball class defines the ball's position, physics, and behavior. It handles the movement and reset functionality.

- Attributes: Stores ball position and Rigidbody.
- Methods: Kick() applies force to the ball, and ResetPosition() resets it.
- Relationship: The Player class interacts with the Ball to perform shots.

Goal Class
The Goal class detects if a goal has been scored when the ball crosses the goal line.

- Attributes: Tracks whether a goal has been scored.
- Methods: DetectGoal() checks if the ball entered the goal.
- Relationship: Works with the GameManager to update the score.\

UIManager Class
The UIManager class handles the game's user interface, such as the pause menu and HUD.

- Attributes: References to the pause menu and game HUD.
- Methods: Controls UI visibility with ShowPauseMenu() and HidePauseMenu().
- Relationship: Works with the GameManager to manage gameplay and UI elements.

Class Relationships Overview
GameManager --> Ball (Controls)

- The GameManager oversees the ball’s behavior, controlling its movement and interactions during gameplay.

GameManager --> Goal (Checks Goal)

- The GameManager checks if the ball has entered the goal to update the score and trigger relevant game events.

Player --> Ball (Shoots)

- The Player class interacts with the Ball to perform shots during gameplay.

UIManager --> GameManager (Manages UI)

- The UIManager class interacts with the GameManager to manage UI elements such as the pause menu, HUD, and in-game notifications.

The Class Diagram provides a structured blueprint of the Football Game System, illustrating the relationships and dependencies between different components. It serves as a fundamental reference for developers, ensuring efficient organization of code, modularity, and maintainability. By defining key classes such as GameManager, Player, Ball, Goal, and UIManager, the diagram encapsulates the core logic and interactions required for smooth gameplay.

Key Aspects of the Class Diagram:

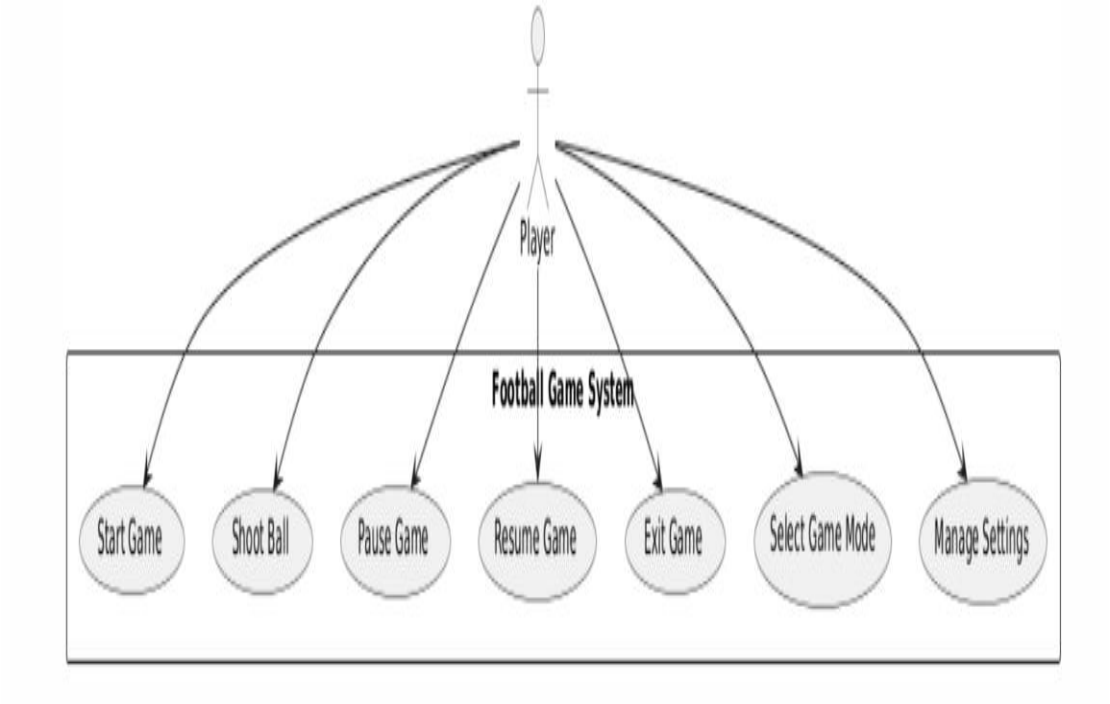
Encapsulation of Core Game Logic:

- The GameManager class acts as the central controller, managing game flow, scoring, and interactions between various game objects.
- It ensures that all other components, such as the Ball, Goal, and UI, function cohesively.

Player Interaction and Ball Control:

- The Player class represents user input and actions, allowing movement and ball control through shooting mechanics.
- The Ball class contains physics properties, including position and velocity, ensuring realistic movement when kicked.

4.3 Usecase Diagram:



The Use Case Diagram visually represents how a Player interacts with the Football Game System, showing different functionalities available to the user. It includes essential actions that a player can perform during gameplay.

Actor in the System:

- Player: The main user who interacts with the game by playing, pausing, changing settings, and exiting.

Use Cases and Their Functions:

Start Game (UC1)

- The player initiates the game from the main menu.
- The system loads the selected game mode and prepares the environment for play.

Shoot Ball (UC2)

- The player takes a shot using the shooting mechanics.
- The game calculates ball trajectory and goal detection.

Pause Game (UC3)

- The player can pause the game at any time.
- The pause menu appears, offering options to resume, manage settings, or exit.

Resume Game (UC4)

- If the game is paused, the player can resume gameplay.
- The pause menu disappears, and the game continues.

Exit Game (UC5)

- The player can choose to exit the game.
- The game session ends, returning the player to the main menu.

Select Game Mode (UC6)

- The player can choose between different modes like penalty shootout, free kick practice, or unlimited mode.
- The system loads the selected mode accordingly.

Manage Settings (UC7)

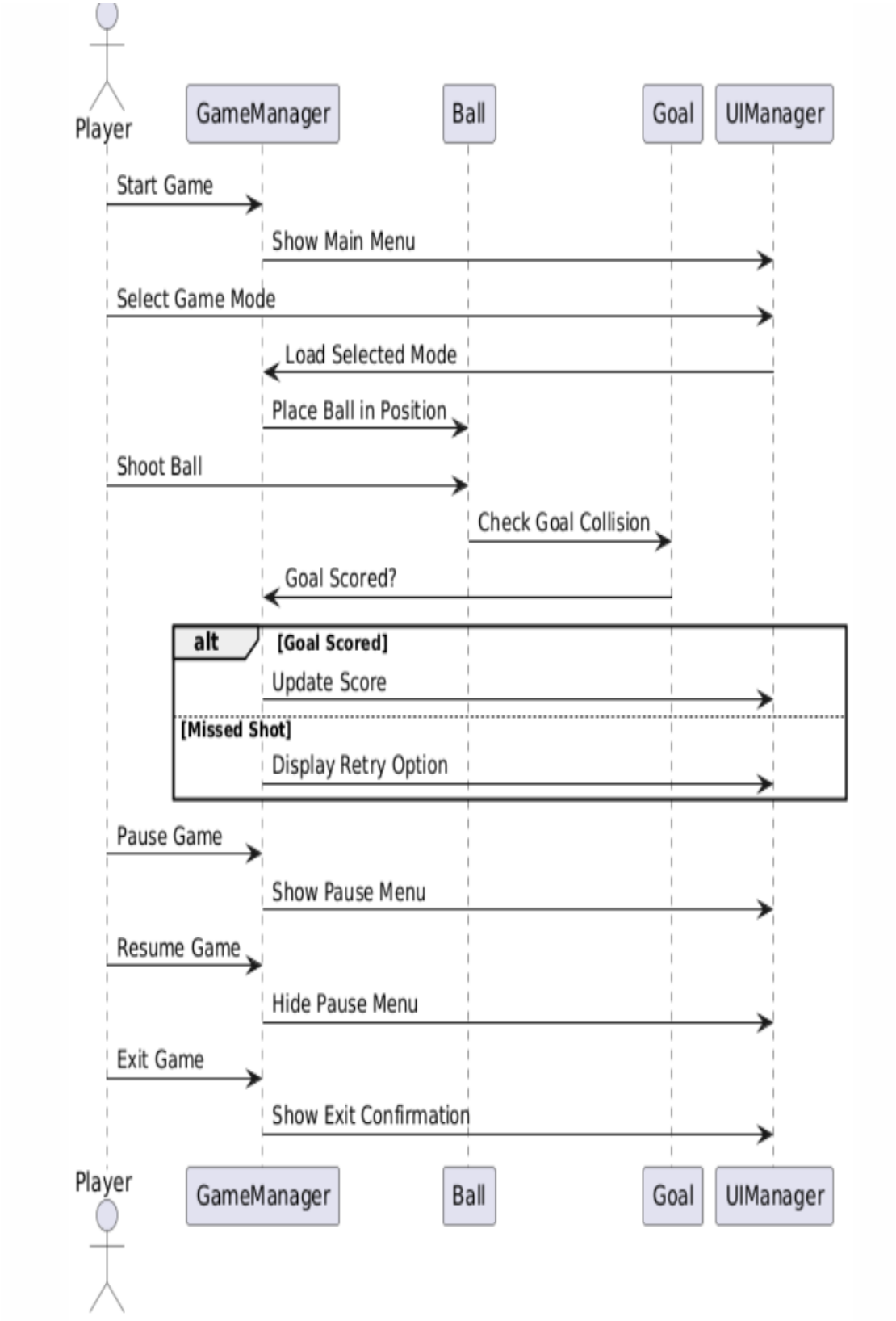
- The player can adjust game settings like sound, graphics, and controls.
- The settings are saved for future sessions.

The Use Case Diagram plays a crucial role in game development by clearly defining the scope of the football game system. It helps developers and designers ensure that all critical functionalities are incorporated and accessible to the player.

- The diagram ensures structured interactions between the player and the game system.
- It aids in identifying potential enhancements, such as adding new game modes or expanding customization features.
- It serves as a foundation for UI/UX design, ensuring an intuitive experience for the player.
- It helps in planning development phases, ensuring that core gameplay elements are prioritized before additional features.

In summary, the Use Case Diagram provides a comprehensive blueprint for the football game's user interactions, ensuring a smooth, enjoyable, and well-structured gameplay experience.

4.4 Sequence Diagram:



Detailed Explanation of the Sequence Diagram for the Football Game System
The Sequence Diagram visually represents the flow of interactions between different components in the Football Game System when a player engages with the game. It illustrates the step-by-step sequence of actions and the way different objects within the system communicate with each other during gameplay. The diagram helps understand the execution flow, system behavior, and how various entities interact dynamically to handle user inputs.

Actors and Participants
The sequence diagram involves multiple actors and system components, each responsible for a specific function in the game. These components collectively ensure smooth gameplay and proper user interaction.
Actors:

1. Player – The user controlling the game, making decisions such as selecting a mode, taking shots, pausing, and navigating through menus.

System Components (Participants):

2. Game Manager (GM) – The central controller that oversees the entire game logic, including game state transitions, score tracking, and handling interactions between game elements.
3. Ball (B) – Represents the football in the game, responding to player input for shooting, moving in accordance with physics, and determining goal collisions.
4. Goal (G) – Detects whether a ball enters the goalpost and informs the Game Manager of a successful goal.
5. UI Manager (UI) – Handles the user interface, managing menus, score updates, pause screens, and game state transitions.

Sequence of Events and System Interactions
Starting the Game

- The Player initiates the game by interacting with the UI to start gameplay.
- The Player sends a request to the GameManager by selecting the "Start Game" option.
- The GameManager instructs the UIManager to display the Main Menu, which provides options such as selecting game modes, adjusting settings, and viewing high scores.
- The Player chooses a game mode, and the UIManager sends this selection back to the GameManager to load the appropriate mode.
- The GameManager initializes the scene and places the Ball at the starting position, preparing for the match to begin.

Shooting the Ball and Checking for a Goal

- The Player controls the shooting mechanics by aiming and pressing the shoot button.
- The Ball moves according to the applied force and direction, following the in-game physics engine.
- As the ball travels, it checks for a collision with objects in the scene, including the Goal.
- If the ball collides with the Goal, the Goal component detects this and immediately notifies the GameManager whether a goal has been scored.
- If no collision is detected, the ball continues moving until it stops, signaling a missed shot.

Handling a Goal or a Missed Shot

- If the ball successfully enters the Goal, the GameManager updates the player's score through the UIManager. The UI displays a confirmation message such as "GOAL!" and updates the score counter.
- The UIManager may also trigger celebratory animations or effects to enhance the player's experience.
- If the shot misses, the UIManager presents a retry option, allowing the Player to attempt the shot again. This maintains engagement by giving players another chance to improve their accuracy.
- The game may also include a limited number of attempts or a countdown timer, adding to the challenge.

Pausing and Resuming the Game

- During gameplay, the Player may press the Pause button to take a break.
- The GameManager receives this input and instructs the UIManager to display the Pause Menu, which provides options such as Resume Game, Go Back to Levels, and Turn Off Music.
- If the Player selects "Resume Game," the GameManager signals the UIManager to close the Pause Menu, and gameplay continues seamlessly.
- If the Player chooses to return to the Levels Menu, the GameManager safely resets the game state before transitioning to the levels selection screen.

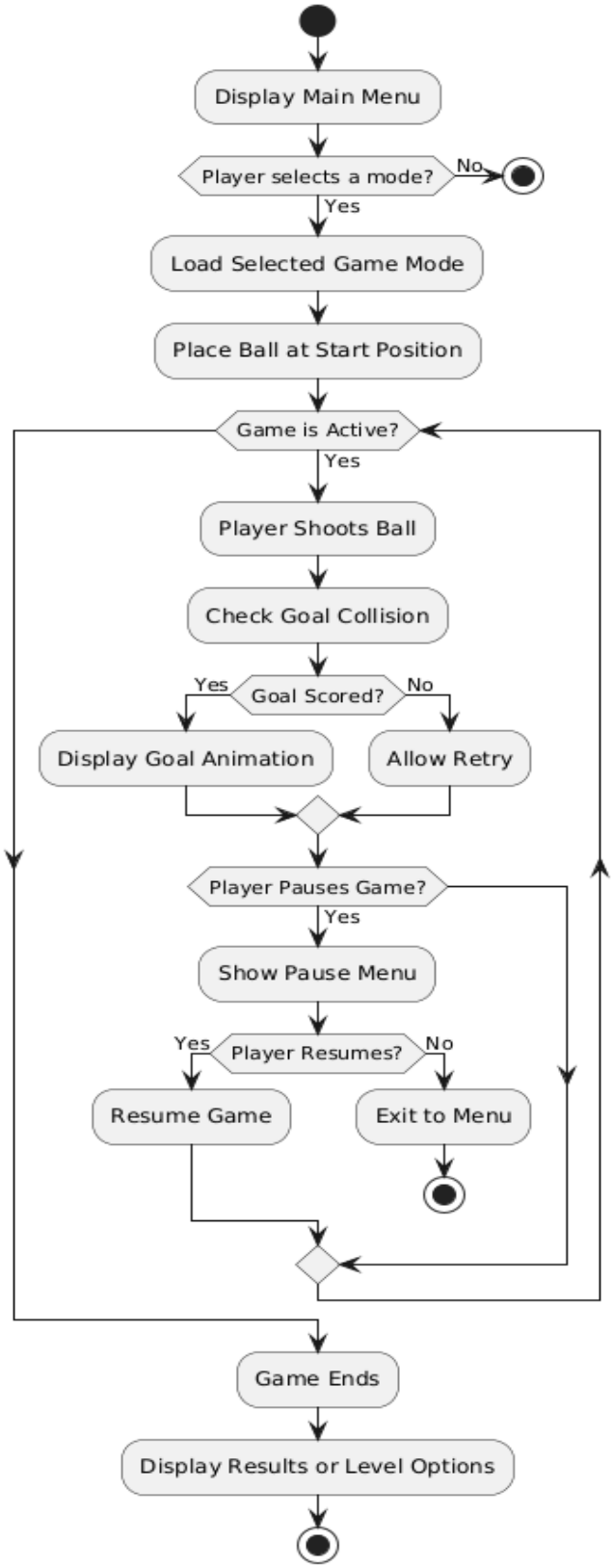
Exiting the Game

- If the Player decides to exit the game, they can do so via the Pause Menu or the Main Menu.
- The GameManager receives this request and asks the UIManager to display an exit confirmation dialog to prevent accidental exits.
- If the Player confirms the exit, the GameManager safely closes the session and returns the player to the main menu or the title screen.
- The GameManager ensures that any unsaved progress or session data is handled before exiting, preventing loss of important information.

The Sequence Diagram of the Football Game System provides a structured overview of how different components interact with each other to deliver a smooth gaming experience. It showcases the game's logical flow, from starting a match to executing shots, scoring goals, pausing gameplay, and eventually exiting.

By implementing this structured sequence of interactions, the Football Game System ensures that player actions are accurately processed, game logic is consistently maintained, and the user interface remains responsive. This design allows for efficient troubleshooting, easy modifications, and future scalability for additional features like multiplayer functionality, advanced AI, or new game modes.

4.5 Activity Diagram:



Detailed Explanation of the Activity Diagram for the Football Game System

The Activity Diagram provides a clear visual representation of the workflow in the Football Game System, depicting the various sequences of activities that occur from

the start of the game to its completion. It outlines player interactions, system responses, and decision points that dictate the flow of gameplay.

Flow of Activities

Displaying the Main Menu

- When the game launches, the Main Menu is displayed, serving as the primary interface for the player.
- The Main Menu presents multiple options, such as:
    - Selecting a game mode
    - Viewing settings
    - Checking high scores
    - Exiting the game
- The player must make a decision:
    - If they select a game mode, the system proceeds to load the chosen mode.
    - If they choose to exit, the game terminates, returning the player to their desktop or home screen.

Loading the Selected Mode

- Upon selecting a game mode, the system initializes the game environment.
- The following key actions take place:
    - The selected level is loaded.
    - The Ball is placed at its starting position.
    - The UI updates to reflect relevant game settings, such as score tracking, time limit, or special objectives.
- Once the environment is set up, the Player gains control and can begin gameplay.

Main Game Loop (Active Gameplay)

- The game enters its main active state, where the player actively controls the ball and interacts with the game environment.
- The player's actions may include:
    - Aiming and shooting the ball toward the goal.
    - Adjusting shot power and angle to increase precision.
    - Attempting to score by avoiding defenders and goalkeepers.
- The system continuously checks for in-game events such as:
    - Collisions (e.g., ball hitting the goal, wall, or obstacles).
    - Time progression (if the mode includes a timer).

Handling Goal or Missed Shot

- When the Player takes a shot, the system evaluates whether the ball has:
    - Entered the Goal (Successful Shot)
    - Missed the Goal (Unsuccessful Shot)

- If the shot is successful, the following actions take place:
    - A goal animation is displayed, and the score counter is updated.
    - The ball is reset to its initial position, and the player prepares for the next attempt.
- If the shot misses, the system presents options to the player:
    - Retry the shot, allowing another attempt.
    - Continue playing, if the game has multiple attempts per round.
    - If applicable, end the session when attempts run out.

Handling the Pause Menu

- At any time during the match, the Player may choose to pause the game.
- Pressing the Pause Button triggers the following actions:
    - The Pause Menu is displayed, overlaying the gameplay.
    - The game freezes (pausing all movement and timers).
- The Pause Menu provides options:
    - Resume Game – Closes the menu and resumes gameplay.
    - Turn Off Music – Toggles background music on or off.
    - Go Back to Levels Menu – Exits the current session and returns to level selection.
- If the Player chooses to exit, the session ends, and progress (if applicable) is saved before transitioning to the Main Menu or Levels Menu.

Game Ending and Session Completion

- The game ends when one of the following occurs:
    - The Player completes all required shots (if in a challenge mode).
    - The timer reaches zero (if in a timed mode).
    - The Player chooses to exit via the Pause Menu.
- Upon session completion, the system displays a results screen, providing:
    - Performance summary (e.g., goals scored, shot accuracy, time remaining).
    - Options to either proceed to the next level or return to level selection.
- The Player is given the option to:
    - Retry the current level, starting fresh.
    - Advance to the next level, loading a more challenging experience.
    - Return to the main menu, exiting gameplay.

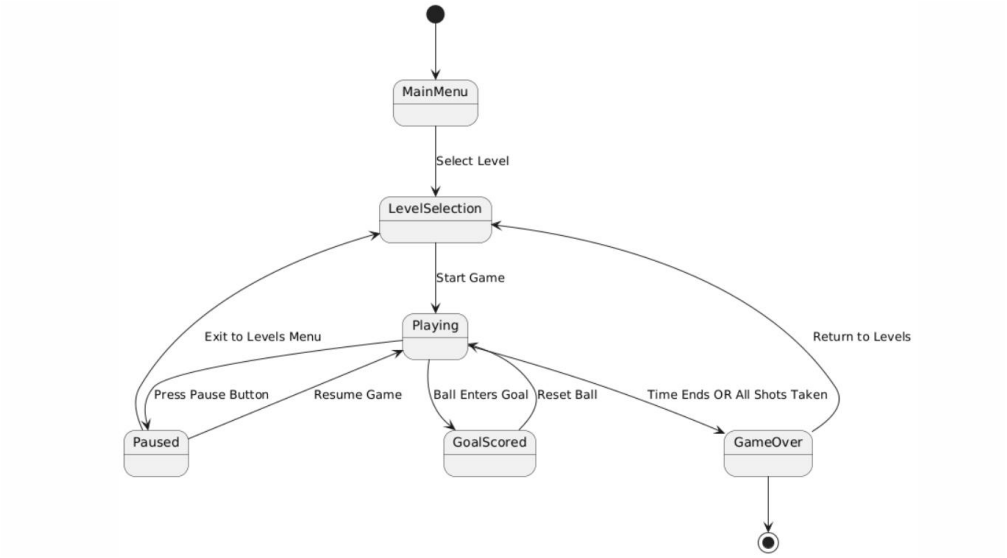
The Activity Diagram provides an efficient and structured workflow for the Football Game System, ensuring smooth gameplay transitions from start to finish. It defines player interactions at every stage, highlights decision points, and outlines how the system responds based on user input.

By following this structured activity flow, the game maintains a seamless experience, ensuring that key functionalities like goal detection, shot mechanics, pause handling,

and session completion operate efficiently. Additionally, this design allows future enhancements such as:

- Adding multiplayer functionality
- Introducing dynamic AI opponents
- Implementing new challenge modes and tournaments

4.6. State Diagram



The State Diagram represents the different states of the Football Game System and how the game transitions between these states based on player actions and system conditions.

States and Transitions:

1Initial State → Main Menu

- The game starts at the Main Menu, where the player can navigate through options.

Main Menu → Level Selection

- The player selects a level or game mode to play, transitioning to the Level Selection screen.

Level Selection → Playing

- Once the player selects a level, the game enters the Playing state, where the player can control the ball and attempt to score goals.

Playing → Paused

- If the player presses the Pause Button, the game moves to the Paused state.
- From here, the player can either resume the game or exit back to the level selection menu.

Playing → Goal Scored

- If the ball enters the goal, the game transitions to the GoalScored state, where it updates the score.
- The system then resets the ball and resumes the game.

Playing → Game Over

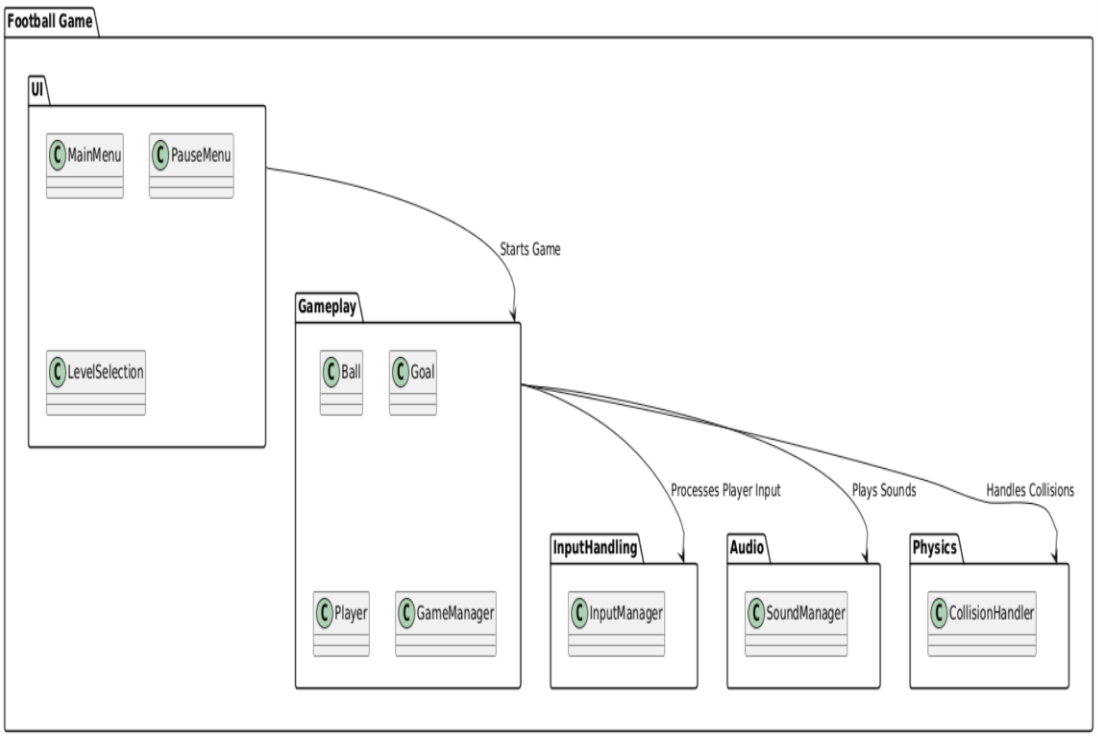
- The game ends when either:
    - The timer runs out (in timed modes).

- All shots have been taken (in limited-attempt modes).
- The game moves to the GameOver state, displaying the results.

Game Over → Level Selection OR End

- The player can either:
  - Return to the Levels menu to choose another level.
  - Exit the game, leading to the final state.

4.7 Package diagram:



The Package Diagram provides a high-level structure of the Football Game System, organizing different components into logical packages based on their functionality.

Packages and Their Roles:

UI Package

- MainMenu → Handles the main menu interface where players can start the game or navigate to other sections.
- PauseMenu → Displays the pause menu when the game is paused, allowing the player to resume or exit.
- LevelSelection → Allows the player to choose a level or game mode before starting.

Gameplay Package

- Ball → Represents the football and its movement in the game.
- Goal → Detects when a goal is scored and updates the score accordingly.
- Player → Manages player interactions like shooting the ball.
- GameManager → Controls the overall game flow, including starting, pausing, and ending matches.

InputHandling Package

- InputManager → Processes player inputs such as movement and shooting.

Audio Package

- SoundManager → Manages sound effects and background music in the game.

Physics Package

- CollisionHandler → Handles interactions between game objects, such as ball-to-goal detection.

Relationships Between Packages:

UI → Gameplay

- The UI components start and control the game (e.g., starting a match from the Main Menu).

Gameplay → InputHandling

- The Gameplay package processes player inputs via the InputManager.

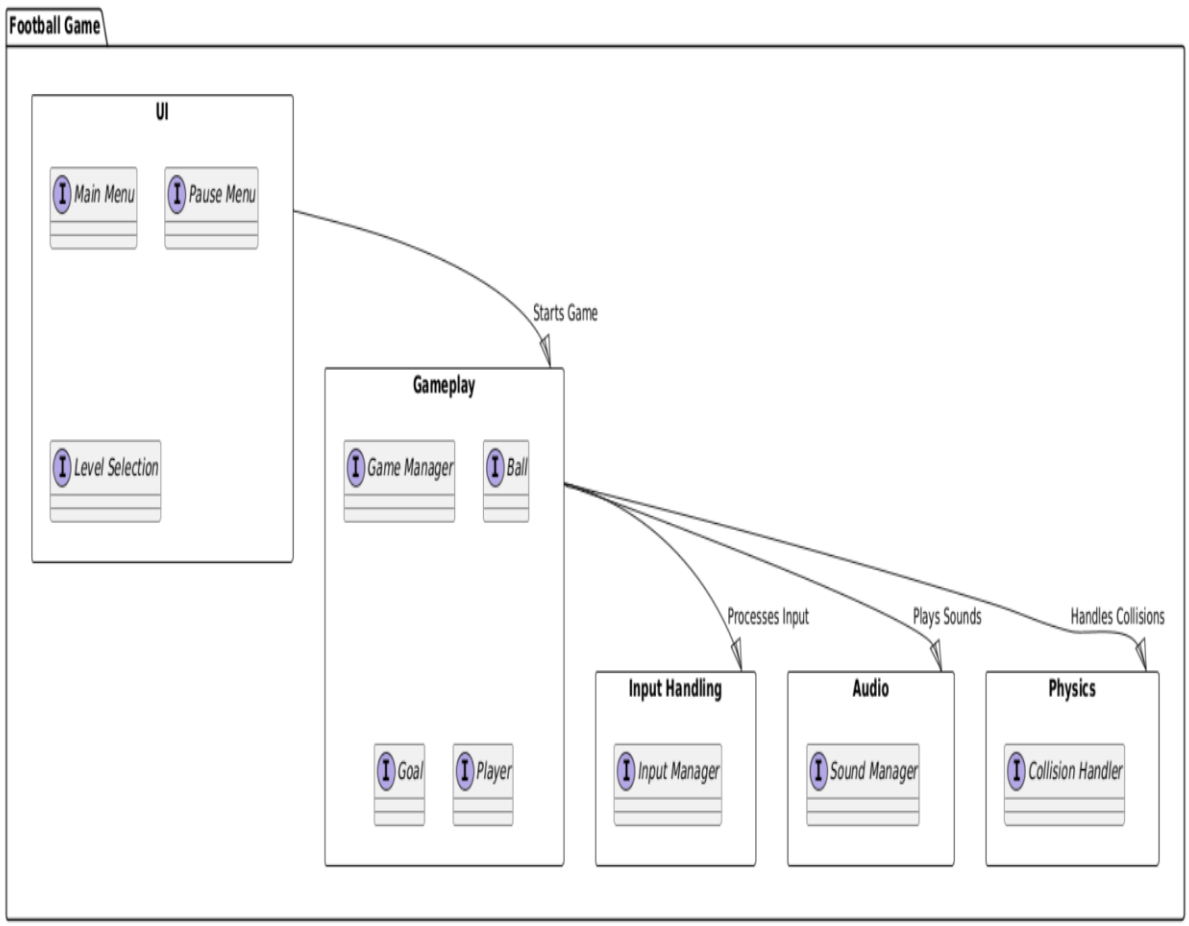
Gameplay → Physics

- The Gameplay mechanics depend on the Physics package for realistic ball movement and collisions.

Gameplay → Audio

- The SoundManager plays sounds when actions occur in the game, like shooting or scoring.

4.8 Component diagram:



The Component Diagram provides a structural overview of how different modules in the Football Game System interact with each other. Each component represents a functional module, and interfaces define how they communicate.

- Key Components and Their Roles:

UI Component (User Interface)

- Main Menu → Handles navigation, starting the game, and accessing settings.
- Pause Menu → Displays game pause options like resuming, quitting, or changing settings.
- Level Selection → Allows the player to choose a game mode or level.

Gameplay Component

- Game Manager → Controls the overall game state, including starting, pausing, and restarting matches.
- Ball → Represents the football and its behavior during gameplay.
- Goal → Detects when a goal is scored and updates the game state.
- Player → Manages player movement and shooting.

Input Handling Component

- Input Manager → Processes user inputs, such as moving, aiming, and shooting.

Audio Component

- Sound Manager → Manages background music and sound effects for interactions like kicks and goals.

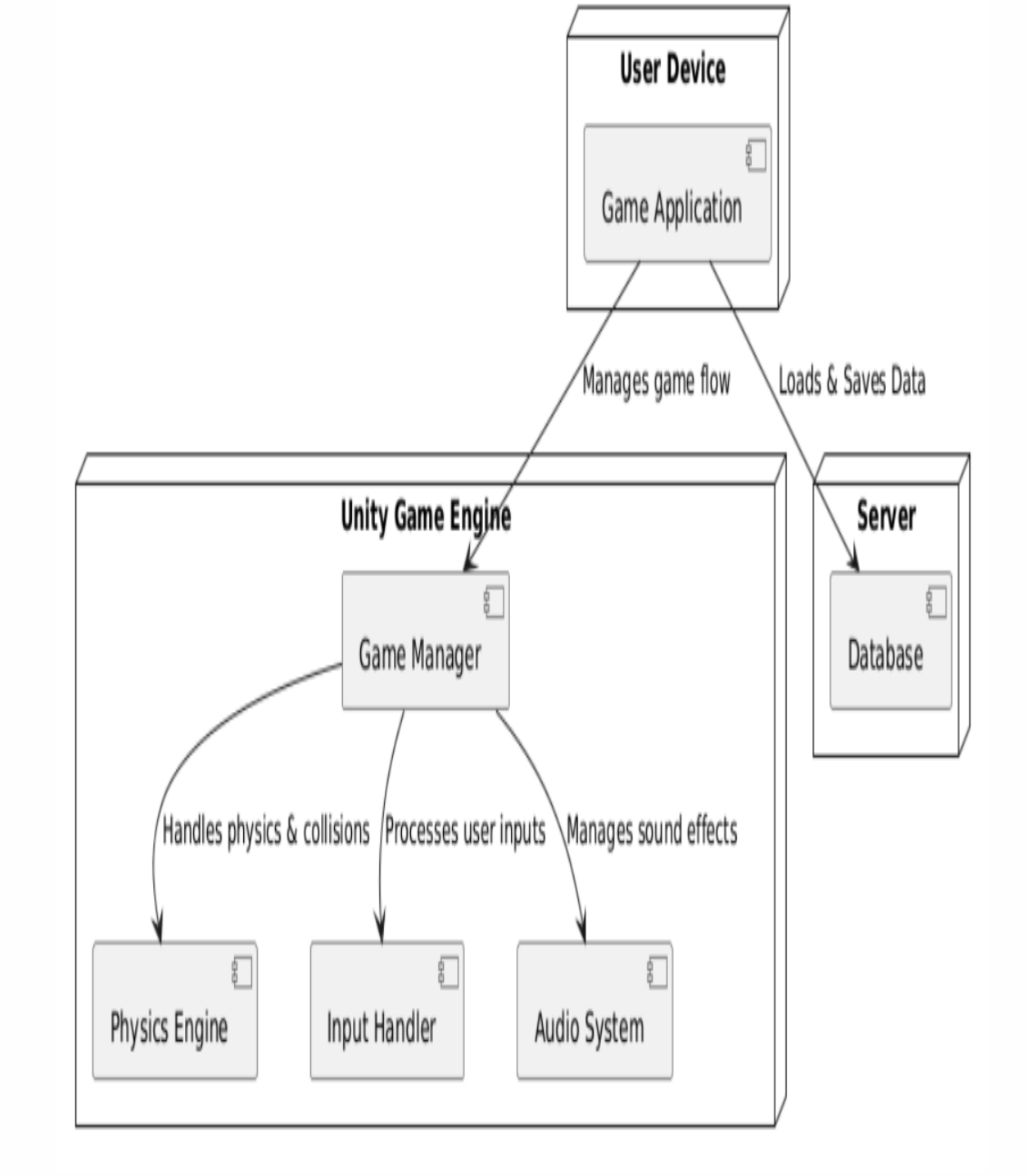
Physics Component

- Collision Handler → Detects interactions between the ball, player, and goal.

Relationships Between Components:

- UI → Gameplay
  - The UI components interact with the Gameplay system to start matches and handle user input.
- Gameplay → Input Handling
  - The Input Manager processes the player's actions and sends commands to the Gameplay component.
- Gameplay → Physics
  - The Physics engine ensures that the ball movement, collisions, and physics-based interactions behave realistically.
- Gameplay → Audio
  - The Sound Manager plays effects when a goal is scored, a shot is taken, or a button is pressed.

4.9 Deployment diagram:



The Deployment Diagram showcases how the Football Game System is structured across different hardware and software components. It highlights how the game runs on a user's device, interacts with the Unity Game Engine, and connects to an external server for data management.

Key Components in Deployment:

User Device (Client)

- Game Application → The main game executable that runs on the player's device.
- Handles game logic, user inputs, rendering, and UI interactions.

Unity Game Engine (Processing Layer)

- Game Manager → Controls the game state, rules, and mechanics.
- Physics Engine → Ensures realistic ball movement, collisions, and gravity effects.
- Input Handler → Processes user inputs (keyboard, gamepad, touchscreen, etc.).
- Audio System → Manages background music, sound effects, and in-game audio feedback.

Server (Data Layer)

- Database → Stores user profiles, settings, and other persistent data.

Relationships Between Components:

Game Application → Game Manager

- The Game Manager controls the game's main logic, such as handling player actions and match progression.

Game Manager → Physics Engine

- The Physics Engine simulates real-world behaviors like ball movement, goal detection, and bounces.

Game Manager → Input Handler

- The Input Handler processes player actions like shooting, movement, and UI navigation.

Game Manager → Audio System

- The Audio System plays sounds for goal scoring, kicks, and button presses.

Game Application → Database

- The Game Application connects to the server to store and retrieve player data, including user preferences and settings.

4.10 Database Design:

-players:

Table structure | Relation view

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	UserName	varchar(11)	utf8_general_ci		No	None			Change Drop More
2	password	varchar(255)	utf8_general_ci		No	None			Change Drop More

Check all | With selected: Browse Change Drop Primary Unique Index Spatial Fulltext

Overview:

The Players Table is a fundamental part of the database structure, responsible for storing user-related information. It primarily manages authentication details, ensuring that players can securely log in and interact with the game. This table plays a crucial role in identifying individual users, maintaining their credentials, and potentially storing additional attributes relevant to gameplay.

Purpose of the Players Table:

The Players Table serves multiple purposes, including but not limited to:

- User Authentication & Login Management:
  - Ensures that each player has unique login credentials.
  - Facilitates secure access to the game by verifying the username and password combination.
- User Identification & Profile Management:
  - Each player is assigned a unique username, which acts as their primary identifier in the system.
  - Password storage ensures users can log in securely while maintaining privacy.
- Data Integrity & Security:
  - Unique constraints on usernames ensure that no two users can have the same identifier.
  - Implementing encryption (such as hashing passwords) ensures that user credentials remain secure and protected from unauthorized access.
- Foundation for Player-Based Features:
  - Serves as a reference for tracking player performance, progress, or any other attributes that may be implemented in future updates.
  - Can be expanded to include additional details such as email addresses, profile pictures, and in-game statistics.

Detailed Explanation of Columns:

username (Primary Key, Unique, Not Null)

- The username is the main identifier for each player in the database.
- It is marked as the Primary Key (PK), ensuring that each player has a unique identity.
- The UNIQUE constraint prevents duplicate usernames from being registered.
- The NOT NULL constraint ensures that every player has a valid username, preventing anonymous access.
- It is typically stored as a VARCHAR(50) to allow flexibility while maintaining efficiency in storage.

password (Not Null, Secure Storage Recommended)

- Stores the player's password used for authentication.
- The NOT NULL constraint ensures that each player must have a valid password.
- Best practice: Instead of storing plain text passwords, the system should use secure hashing algorithms such as bcrypt, Argon2, or PBKDF2 to encrypt the passwords before storing them in the database.
- Passwords are stored as VARCHAR(255) to accommodate hashed values, which tend to be longer than regular text passwords.

Conclusion:

The Players Table is a critical component of the database, serving as the primary repository for user authentication and identification. With its structured approach to

managing usernames and passwords, it ensures a secure and efficient login system for the football game.
By implementing password hashing, unique constraints, and security measures, the table safeguards user data while maintaining scalability for future enhancements. Additional fields such as email, last login, and profile picture could further enhance user engagement, allowing for better account management and security features.
This foundational table ensures that every player has a unique identity in the system while providing a secure and scalable environment for future developments in the game.

5.System Implementation:

C# Files:

5.1.FreekickCore.cs:

```
using System.Collections;
using TMPro;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class FreekickCore : MonoBehaviour
{
    [Header("Ball and Football Arena")]
    public Rigidbody ball;
    public AudioSource shootAudioSource;
    public AudioClip shootAudioClip;
    public GameObject footballArena;

    [Header("UI")]
    public RectTransform ballHitPos;
    public Slider shootSlider;
    public TrajectoryLineRenderer trajectoryLineRenderer;
    public GameObject ballHitCanvas, directionCanvas,
freekickPositionSelectionPanel, freekickSettingPanel, freekickCompletedPanel;
    public TextMeshProUGUI ballDistanceText;

    [Header("Points")]
    public Transform freekickPoint;

    public Transform penaltySpot;

    public Transform farTarget, nearTarget, goalCenter;
```

```
    private float _power;
    private const float XMin = -34.8f, XMax = 34.8f, ZMin = -103f, ZMax = 0,
BallHitPosRadius = 0.435f;
    private Vector3 _farTargetLocalPos, _nearTargetLocalPos;
    private bool _isUnavailableForNewFreekick, _isShooting, _isBallHit,
_isSetShootSlider;
    private bool _isRightKickAngle, _isLeftKickAngle;
    private bool _isForwardFreekickPosition, _isBackFreekickPosition,
_isRightFreekickPosition, _isLeftFreekickPosition;
    private bool _isUpBallHitPos, _isDownBallHitPos, _isRightBallHitPos,
_isLeftBallHitPos;

    private void Awake()
    {
        _farTargetLocalPos = farTarget.localPosition;
        _nearTargetLocalPos = nearTarget.localPosition;
    }


private void Update()
{
    // Check if the active scene is "penald3" or "penald4"
    bool isSceneRestricted = SceneManager.GetActiveScene().name == "penald2" ||
SceneManager.GetActiveScene().name == "penald3" ||
SceneManager.GetActiveScene().name == "penald4" ||
SceneManager.GetActiveScene().name == "penald5";

    // Opens and closes "Football Arena" gameObject if pressing "F" button.
    if(Input.GetKeyDown(KeyCode.F))
        footballArena.SetActive(!footballArena.activeSelf);

    // When available, creates new freekick if pressing "Enter" button.
    if (!_isUnavailableForNewFreekick && Input.GetKeyDown(KeyCode.Return))
        NewFreekickCreate();

    // Set Kick Angles.
    _isRightKickAngle = !_isBallHit && Input.GetKey(KeyCode.RightArrow);
    _isLeftKickAngle = !_isBallHit && Input.GetKey(KeyCode.LeftArrow);

    // Set Freekick Position (Disable WASD movement in restricted scenes)
    if (!isSceneRestricted)
    {
        _isForwardFreekickPosition = !_isShooting && Input.GetKey(KeyCode.W);
        _isBackFreekickPosition = !_isShooting && Input.GetKey(KeyCode.S);
        _isRightFreekickPosition = !_isShooting && Input.GetKey(KeyCode.D);
        _isLeftFreekickPosition = !_isShooting && Input.GetKey(KeyCode.A);
    }
    else
    {
```

```
        _isForwardFreekickPosition = false;
        _isBackFreekickPosition = false;
        _isRightFreekickPosition = false;
        _isLeftFreekickPosition = false;
    }
    if (_isShooting)
    {
        if (directionCanvas.activeSelf) directionCanvas.SetActive(false);
        if (!_isBallHit)
        {
            // Applying UI changes for shooting.
            if (!ballHitCanvas.activeSelf) ballHitCanvas.SetActive(true);
            if (!trajectoryLineRenderer.gameObject.activeSelf)
trajectoryLineRenderer.gameObject.SetActive(true);
            SetInstructionPanels(false, true, false);

            // Set New Freekick Shoot Ball Hit Pos (curve).
            _isUpBallHitPos = Input.GetKey(KeyCode.W);
            _isDownBallHitPos = Input.GetKey(KeyCode.S);
            _isRightBallHitPos = Input.GetKey(KeyCode.D);
            _isLeftBallHitPos = Input.GetKey(KeyCode.A);

            var originPoint = Vector3.zero;
            var ballHitPosLocal = ballHitPos.localPosition;
            var distance = Vector3.Distance(ballHitPosLocal, originPoint);
            var fromOriginToObject = ballHitPosLocal - originPoint;

            if (distance > BallHitPosRadius)
            {
                fromOriginToObject *= BallHitPosRadius / distance;
                ballHitPos.localPosition = originPoint + fromOriginToObject;
            }

            // Set Freekick Curve Settings.
            ballHitPosLocal = ballHitPos.localPosition;
            farTarget.localPosition = new Vector3(_farTargetLocalPos.x -
ballHitPosLocal.x * 20f, _farTargetLocalPos.y - ballHitPosLocal.y * 5f,
_farTargetLocalPos.z);
            nearTarget.localPosition = new Vector3(_nearTargetLocalPos.x -
ballHitPosLocal.x * 0.5f, _nearTargetLocalPos.y - ballHitPosLocal.y * 0.5f,
_nearTargetLocalPos.z);
            trajectoryLineRenderer.CreateTrajectoryLine(ball.transform, nearTarget,
farTarget);
            trajectoryLineRenderer.transform.position = Vector3.zero;

            // Set slider and shoot freekick.
            _isSetShootSlider = Input.GetKey(KeyCode.Space);

            // Apply force to ball for shooting.
            if (Input.GetKeyUp(KeyCode.Space))
```

```
            {
                ShootToTargets();
                StartCoroutine(WaitForResetShootSlider());
            }
        }
        else
        {
            // Set Trajectory lines.
            if (ballHitCanvas.activeSelf) ballHitCanvas.SetActive(false);
            if (trajectoryLineRenderer.gameObject.activeSelf)
trajectoryLineRenderer.gameObject.SetActive(false);
        }
    }
    else SetNewFreekickUI();
}

    private void FixedUpdate()
    {
        // Applying Kick Angles Input.
        if(_isRightKickAngle) SetKickAngle(1);
        if(_isLeftKickAngle) SetKickAngle(-1);

        // Applying Freekick Position Input.
        if(_isForwardFreekickPosition)
TranslateFreekickPointWithInput(Vector3.forward);
        if(_isBackFreekickPosition) TranslateFreekickPointWithInput(-
Vector3.forward);
        if(_isRightFreekickPosition) TranslateFreekickPointWithInput(Vector3.right);
        if(_isLeftFreekickPosition) TranslateFreekickPointWithInput(-Vector3.right);

        // Applying Freekick Ball Hit Pos Input.
        if (_isUpBallHitPos || _isDownBallHitPos || _isRightBallHitPos ||
_isLeftBallHitPos)
        {
            var ballHitDiff = ballHitCanvas.transform.localScale.magnitude;
            if (_isUpBallHitPos) TranslateBallHitPosWithInput(Vector3.up, ballHitDiff);
            if(_isDownBallHitPos) TranslateBallHitPosWithInput(-Vector3.up,
ballHitDiff);
            if(_isRightBallHitPos) TranslateBallHitPosWithInput(Vector3.right,
ballHitDiff);
            if(_isLeftBallHitPos) TranslateBallHitPosWithInput(-Vector3.right,
ballHitDiff);
        }

        // Applying Shoot Slider Input.
        if(_isSetShootSlider) SetShootSlider();
    }

    private void NewFreekickCreate()
    {
```

```
        // When creating new freekick, all scene components and booleans are setting
here.
        if (trajectoryLineRenderer.gameObject.activeSelf)
trajectoryLineRenderer.gameObject.SetActive(false);
        _isBallHit = false;
        _isShooting = !_isShooting;
        ball.Sleep();
        var ballTransform = ball.transform;
        ballTransform.localPosition = Vector3.zero;
        ballTransform.localEulerAngles = Vector3.zero;
    }

    private void SetNewFreekickUI()
    {
        // Sets all UI components for new freekick.
        if (!directionCanvas.activeSelf) directionCanvas.SetActive(true);
        if (ballHitCanvas.activeSelf) ballHitCanvas.SetActive(false);
        if (!ballDistanceText.gameObject.activeSelf)
ballDistanceText.gameObject.SetActive(true);

        ballDistanceText.text = DistanceConversion.Distance((ball.position -
goalCenter.position).magnitude);
        SetInstructionPanels(true, false, false);
    }

    private void TranslateFreekickPointWithInput(Vector3 direction)
    {
        freekickPoint.Translate(direction * (12f * Time.fixedDeltaTime));

        // restricting the ball in an area.
        if (freekickPoint.transform.localPosition.x > XMax ||
freekickPoint.transform.localPosition.x < XMin ||
            freekickPoint.transform.localPosition.z > ZMax ||
freekickPoint.transform.localPosition.z < ZMin)
        {
            freekickPoint.Translate(-direction * (12f * Time.fixedDeltaTime));
        }
    }

    private void TranslateBallHitPosWithInput(Vector3 direction, float ballHitDiff)
    {
        // Sets Ball Hit Position.
        ballHitPos.Translate(direction * (ballHitDiff * 0.2f * Time.fixedDeltaTime));
    }

    private void SetKickAngle(int direction)
    {
        // Rotates the freekick point with kick angles.
        freekickPoint.transform.Rotate(freekickPoint.up, direction * 0.6f);
        trajectoryLineRenderer.transform.eulerAngles = Vector3.zero;
```

```
    }

    private void SetInstructionPanels(bool positionSelection, bool setting, bool
completed)
    {
        // Set UI visibilities.
        if (freekickPositionSelectionPanel.activeSelf != positionSelection)
freekickPositionSelectionPanel.SetActive(positionSelection);
        if (freekickSettingPanel.activeSelf != setting)
freekickSettingPanel.SetActive(setting);
        if (freekickCompletedPanel.activeSelf != completed)
freekickCompletedPanel.SetActive(completed);
    }

    private void SetShootSlider()
    {
        // Set shoot slider if player presses "Space" key.
        if (_power >= 30f) return;
        _power++;
        shootSlider.value = _power / 30f;
    }

    private void ResetShootSlider()
    {
        // Resets power, shoot slider and getting available for new freekick.
        _power = 0;
        shootSlider.value = 0;
        _isUnavailableForNewFreekick = false;
    }

    private IEnumerator WaitForResetShootSlider()
    {
        // Waits seconds after shooting for avoid the possibility of the player accidentally
skipping the shoot and taking a new freekick.
        _isBallHit = true;
        _isUnavailableForNewFreekick = true;
        yield return new WaitForSecondsRealtime(4f);
        SetInstructionPanels(false, false, true);
        ResetShootSlider();
    }

    private void ShootToTargets()
    {
        // Shooting near and far targets.
        if (ballDistanceText.gameObject.activeSelf)
ballDistanceText.gameObject.SetActive(false);
        SetInstructionPanels(false, false, false);
        BallSoundPlayer.PlaySound(shootAudioSource, shootAudioClip, 1 + (2 -
_power * 0.0666f));
        StartCoroutine(ShootToNearTarget());
```

```
    }

    private IEnumerator ShootToNearTarget()
    {
        // Shooting near target for initial movement.
        while ((ball.position - nearTarget.position).magnitude > 0.2f)
        {
            ball.position = Vector3.MoveTowards(ball.position, nearTarget.position, 0.1f
+ _power / 60f);
            yield return new WaitForSeconds(0.01f);
        }

        ShootToFarTarget();
        yield return null;
    }

    private void ShootToFarTarget()
    {
        // Adding force if ball arrives the near target.
        var shoot = (farTarget.position - ball.position).normalized;
        ball.AddForce((shoot + new Vector3(0f, _power / 105f, 0f)) * _power / 2.4f,
ForceMode.Impulse);
    }
}
```

5.2.BallCollisionHandler.cs:

```
using UnityEngine;


public class BallCollisionHandler : MonoBehaviour

{

    private Rigidbody ball;


    [Header("AudioSources")]

    public AudioSource netAudioSource;

    public AudioSource crossbarAudioSource;

    public AudioSource ballBounceAudioSource;


    [Header("AudioClips")]
```

```
    public AudioClip netAudioClip;
    public AudioClip crossbarAudioClip;
    public AudioClip ballBounceAudioClip;

    private void Awake()
    {
        ball = GetComponent<Rigidbody>();
    }

    private void OnCollisionEnter(Collision collision)
    {
        var ballVelocity = ball.velocity.magnitude;

        if (collision.gameObject.layer == LayerMask.NameToLayer($"Ball Limit
Colliders"))
        {
            BallSoundPlayer.PlaySoundByVelocity(netAudioSource, netAudioClip, 1 +
12f / ballVelocity, ballVelocity, 3f);

            while (ball.velocity.magnitude > 3f)
            {
                ball.velocity *= 0.9f;
            }
        }

        if (collision.gameObject.layer == LayerMask.NameToLayer($"Crossbar
Collider"))
        {
```

```
            BallSoundPlayer.PlaySoundByVelocity(crossbarAudioSource,
crossbarAudioClip, 1 + 5f / ballVelocity, ballVelocity, 1f);
        }


        if (collision.gameObject.layer == LayerMask.NameToLayer($"Grass"))
        {
            BallSoundPlayer.PlaySoundByVelocity(ballBounceAudioSource,
ballBounceAudioClip, 1 + 3f / ballVelocity, ballVelocity, 1f);
        }
    }
}
```

5.3.GameManager.cs:

```
using Assets.SuperGoalie.Scripts.Entities;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Idle.MainState;
using Patterns.Singleton;
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
using UnityEngine.Networking;

namespace Assets.SuperGoalie.Scripts.Managers
{
    public class GameManager : Singleton<GameManager>
    {
```

```
[SerializeField] float _ballDribbleForce = 5f;
[SerializeField] float _ballKickForce = 15;
[SerializeField] Ball _ball;
[SerializeField] Goal _goal;
[SerializeField] GoalKeeper _goalKeeper;
[SerializeField] Text _scoreText;
[SerializeField] Text _timerText; // Timer UI Text
[SerializeField] GameObject gameOverPanel; // Game Over Panel

bool _run = true;
int _score;
Vector3 _ballInitPos;
Quaternion _ballInitRot;

protected Transform Cam;
protected Vector3 CamForward;

bool _isScene2 = false; // Check if this is Scene 2 (penald2)
float _timeRemaining = 60f;
bool _timerRunning = false;

public delegate void BallLaunch(float power, Vector3 target);
public BallLaunch OnBallLaunch;

public override void Awake()
{
   // Register events
```

```
        _ball.OnBallLaunched += SoundManager.Instance.PlayBallKickedSound;
        _goalKeeper.OnPunchBall += SoundManager.Instance.PlayBallKickedSound;
        _goal.GoalTrigger.OnCollidedWithBall +=
SoundManager.Instance.PlayGoalScoredSound;

        _ball.OnBallLaunched += _goalKeeper.Instance_OnBallLaunched;
        _goal.GoalTrigger.OnCollidedWithBall +=
Instance_OnBallCollidedWithGoal;
        OnBallLaunch += _ball.Instance_OnBallLaunch;

        // Cache initial ball position
        _ballInitPos = _ball.Position;
        _ballInitRot = _ball.Rotation;

        if (Camera.main != null)
            Cam = Camera.main.transform;
        else
            Debug.LogWarning("Warning: no main camera found.");

        // Check if this is Scene 2 ("penald2") before enabling the timer
        if (SceneManager.GetActiveScene().name == "penald2")
        {
            _isScene2 = true;
            _timerRunning = true;
        }
    }

    private void Instance_OnBallCollidedWithGoal()
```

```
{
    ++_score;
    _scoreText.text = string.Format("Score: {0}", _score);
}

private void Update()
{
    if (!_run)
        return;

    // **Handle Timer (Only in Scene 2)**
    if (_isScene2 && _timerRunning)
    {
        if (_timeRemaining > 0)
        {
            _timeRemaining -= Time.deltaTime;
            _timerText.text = $"Time: {Mathf.CeilToInt(_timeRemaining)}";
        }
        else
        {
            _timeRemaining = 0;
            _timerRunning = false;
            EndGame();
        }
    }

    #region TriggerShooting
```

```
if (Input.GetMouseButtonDown(0))
{
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit))
    {
        _run = false;
        Vector3 target = hit.point;
        OnBallLaunch?.Invoke(_ballKickForce, target);
        StartCoroutine(Reset());
    }
}
else
{
    float horizontalRot = Input.GetAxisRaw("Horizontal");
    float verticalRot = Input.GetAxisRaw("Vertical");
    verticalRot = 0f;

    Vector3 input = new Vector3(horizontalRot, 0f, verticalRot);

    if (Input.anyKeyDown)
    {
        Vector3 Movement = Vector3.zero;

        if (Cam != null)
```

```
            {
                CamForward = Vector3.Scale(Cam.forward, new Vector3(1, 0,
1)).normalized;
                Movement = input.z * CamForward + input.x * Cam.right;
            }
            else
            {
                Movement = input.z * Vector3.forward + input.x * Vector3.right;
            }

            Movement.y = 0.03f;
            _ball.Rigidbody.velocity = Movement * _ballDribbleForce;
        }
    }

    #endregion
}

private void EndGame()
{
    _run = false;
    gameOverPanel.SetActive(true);
    StartCoroutine(SaveScoreToDatabase(_score));
}

private IEnumerator SaveScoreToDatabase(int score)
{
```

```
        string username = DBManager.username;
        WWWForm form = new WWWForm();
        form.AddField("username", username);
        form.AddField("score", score);

        UnityWebRequest www = UnityWebRequest.Post("http://localhost/Unity-Football-Freekick-Game-main/save_score.php", form);
        yield return www.SendWebRequest();

        if (www.result != UnityWebRequest.Result.Success)
        {
            Debug.LogError("Failed to save score: " + www.error);
        }
        else
        {
            Debug.Log("Score saved successfully!");
        }
    }

    private IEnumerator Reset()
    {
        yield return new WaitForSeconds(5f);

        _ball.gameObject.SetActive(false);
        _ball.Stop();
        _ball.Position = _ballInitPos;
        _ball.Rotation = _ballInitRot;
```

```
            _goalKeeper.FSM.ChangeState<IdleMainState>();

            yield return new WaitForSeconds(1f);

            _run = true;
            _ball.gameObject.SetActive(true);
            _goal.GoalTrigger.gameObject.SetActive(true);
        }
    }
}
```

5.4.CameraController.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

namespace Assets.SoccerGameEngine_Basic_.Scripts.Controllers
{
    public class CameraController : MonoBehaviour
    {
```

```
[SerializeField]
float _distanceMaxDisplacement = 30f;

[SerializeField]
float _speedFollow = 3f;

[SerializeField]
Transform target;

[SerializeField]
float distanceMaxZ;
[SerializeField]
float distanceMinZ;

private void Awake()
{
    distanceMaxZ = transform.position.z + _distanceMaxDisplacement;
    distanceMinZ = transform.position.z - _distanceMaxDisplacement;
}

private void LateUpdate()
{
    // find the next position to move
    Vector3 nextPosition = Vector3.MoveTowards(transform.position,
        target.position,
        _speedFollow);
```

```
            // clean the psotion
            nextPosition.x = transform.position.x;
            nextPosition.y = transform.position.y;
            nextPosition.z = Mathf.Clamp(nextPosition.z, distanceMinZ, distanceMaxZ);

            // set the next position
            transform.position = nextPosition;
        }
    }
}
```

5.5.Login.cs:

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.Networking;
using UnityEngine.SceneManagement;

public class Login : MonoBehaviour
{
    public InputField nameField;
    public InputField PasswordField;
    public Button submitButton;

    void Start()
```

```
{
    // Disable submit button initially
    submitButton.interactable = false;

    // Attach VerifyInput to field changes
    nameField.onValueChanged.AddListener(delegate { VerifyInput(); });
    PasswordField.onValueChanged.AddListener(delegate { VerifyInput(); });
}

public void CallLogin()
{
    if (submitButton.interactable) // Check if button is interactable before logging in
    {
        StartCoroutine(LoginPlayer());
    }
}

IEnumerator LoginPlayer()
{
    WWWForm form = new WWWForm();
    form.AddField("name", nameField.text);
    form.AddField("password", PasswordField.text);

    using (UnityWebRequest www = UnityWebRequest.Post("http://localhost/Unity-Football-Freekick-Game-main/login.php", form))
    {
        yield return www.SendWebRequest();
```

```
        if (www.result == UnityWebRequest.Result.Success)
        {
            string response = www.downloadHandler.text;
            Debug.Log("Login Response: " + response); // Debugging line

            if (!string.IsNullOrEmpty(response) && response[0] == '0')
            {
                DBManager.username = nameField.text; // Store username
                Debug.Log("Login Successful! Username: " + DBManager.username);

                SceneManager.LoadScene("startmenu");
            }
            else
            {
                Debug.LogError("Invalid username or password");
            }
        }
        else
        {
            Debug.LogError("Login request failed: " + www.error);
        }
    }
}

public void VerifyInput()
{
```

```
        submitButton.interactable = (!string.IsNullOrEmpty(nameField.text) &&
nameField.text.Length >= 8 &&
                                    !string.IsNullOrEmpty(PasswordField.text) &&
PasswordField.text.Length >= 8);

    }

}
```

5.6.Registration.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
public class Registration : MonoBehaviour
{
   public InputField nameField;
   public InputField PasswordField;

   public Button submitButton;

   public void CallRegister()
   {
    StartCoroutine(Register());
   }

  IEnumerator Register()
{
    WWWForm form = new WWWForm();
    form.AddField("name", nameField.text);
    form.AddField("password", PasswordField.text);

    WWW www = new WWW("http://localhost/Unity-Football-Freekick-Game-
main/register.php", form);
    yield return www;

    // Debug the server response
    Debug.Log("Server Response: " + www.text);

    if (www.text.Trim() == "0")  // Trim to remove unwanted spaces/newlines
    {
        Debug.Log("Registration Success");
        UnityEngine.SceneManagement.SceneManager.LoadScene(0);
    }
    else
    {
```

```
        Debug.Log("Registration Failed. Error # " + www.text);
    }
}

        public void VerifyInput()
    {
         submitButton.interactable = (nameField.text.Length >= 8 &&
PasswordField.text.Length >= 8);
    }


}
```

5.7.Ball.cs:

```
using System;
using UnityEngine;


namespace Assets.SuperGoalie.Scripts.Entities
{
    [RequireComponent(typeof(Rigidbody))]
    [RequireComponent(typeof(SphereCollider))]
    public class Ball : MonoBehaviour
    {
        [Tooltip("The gravity acting on the ball")]
        public float gravity = 9f;


        public delegate void BallLaunched(float flightTime, float velocity, Vector3
initial, Vector3 target);
        public BallLaunched OnBallLaunched;


        public Rigidbody Rigidbody { get; set; }
```

```
    public SphereCollider SphereCollider { get; set; }

    private void Awake()
    {
        //get the components
        Rigidbody = GetComponent<Rigidbody>();
        SphereCollider = GetComponent<SphereCollider>();

        // set the gravity of the ball
        Physics.gravity = new Vector3(0f, -gravity, -0f);
    }

    public void Stop()
    {
        Rigidbody.angularVelocity = Vector3.zero;
        Rigidbody.velocity = Vector3.zero;
    }

    public Vector3 FuturePosition(float time)
    {
        //get the velocities
        Vector3 velocity = Rigidbody.velocity;
        Vector3 velocityXZ = velocity;
        velocityXZ.y = 0f;

        //find the future position on the different axis
        float futurePositionY = Position.y + (velocity.y * time + 0.5f * -gravity *
Mathf.Pow(time, 2));
        Vector3 futurePositionXZ = Vector3.zero;

        //get the ball future position
```

```
        futurePositionXZ = Position + velocityXZ.normalized *
velocityXZ.magnitude * time;

        //bundle the future positions to together
        Vector3 futurePosition = futurePositionXZ;
        futurePosition.y = futurePositionY;

        //return the future position
        return futurePosition;
    }

    public void Launch(float power, Vector3 final)
    {
        //set the initial position
        Vector3 initial = Position;

        //find the direction vectors
        Vector3 toTarget = final - initial;
        Vector3 toTargetXZ = toTarget;
        toTargetXZ.y = 0;

        //find the time to target
        float time = toTargetXZ.magnitude / power;

        // calculate starting speeds for xz and y. Physics forumulase deltaX = v0 * t +
1/2 * a * t * t
        // where a is "-gravity" but only on the y plane, and a is 0 in xz plane.
        // so xz = v0xz * t => v0xz = xz / t
        // and y = v0y * t - 1/2 * gravity * t * t => v0y * t = y + 1/2 * gravity * t * t =>
v0y = y / t + 1/2 * gravity * t
        toTargetXZ = toTargetXZ.normalized * toTargetXZ.magnitude / time;
```

```
    //set the y-velocity
    Vector3 velocity = toTargetXZ;
    velocity.y = toTarget.y / time + (0.5f * gravity * time);

    //return the velocity
    Rigidbody.velocity = velocity;

    //invoke the ball launched event
    BallLaunched temp = OnBallLaunched;
    if (temp != null)
        temp.Invoke(time, power, initial, final);
}

public void Instance_OnBallLaunch(float power, Vector3 target)
{
    //launch the ball
    Launch(power, target);
}

public Quaternion Rotation
{
    get
    {
        return transform.rotation;
    }

    set
    {
        transform.rotation = value;
```

```
        }
    }

    public Vector3 Position
    {
        get
        {
            return transform.position;
        }

        set
        {
            transform.position = value;
        }
    }
  }
}
```

5.8.GoalKeeper.cs:

```
using Assets.SimpleSteering.Scripts.Movement;
using Assets.SuperGoalie.Scripts.FSMs;
using System;
using UnityEngine;

namespace Assets.SuperGoalie.Scripts.Entities
{
    [RequireComponent(typeof(GoalKeeperFSM))]
    [RequireComponent(typeof(RPGMovement))]
    public class GoalKeeper : MonoBehaviour
    {
        /// <summary>
        /// A reference to the dive speed of this instance
        /// </summary>
        [SerializeField]
        float _diveSpeed = 4f;

        /// <summary>
        /// A refernce to the goal keeping of this instance
        /// </summary>
        [SerializeField]
        float _goalKeeping = 0.85f;

        /// <summary>
        /// A reference to the height of this instance
        /// </summary>
```

```
float _height = 1.9f;

/// <summary>
/// A refernce to the jump distance of this instance
/// </summary>
[SerializeField]
float _jumpDistance = 1;

/// <summary>
/// A reference to the jump height of this instance
/// </summary>
[SerializeField]
float _jumpHeight = 0.5f;

/// <summary>
/// A refernce to the goal keeping of this instance
/// </summary>
[SerializeField]
float _reach = 0.5f;

/// <summary>
///  reference to the tend goal distance of this instance
/// </summary>
[SerializeField]
float _tendGoalDistance = 3f;

/// <summary>
```

```
///  reference to the tend goal speed of this instance
/// </summary>
[SerializeField]
float _tendGoalSpeed = 3f;

/// <summary>
/// A reference to this instance's animator
/// </summary>
[SerializeField]
Animator _animator;

/// <summary>
/// A reference to the ball instance
/// </summary>
[SerializeField]
Ball _ball;

/// <summary>
/// A reference to the goal instance
/// </summary>
[SerializeField]
Goal _goal;

/// <summary>
/// A reference to the model root
/// </summary>
[SerializeField]
```

```
    Transform _modelRoot;

    public Action OnHasNoBall;

    public Action OnHasBall;

    public Action OnPunchBall;

    public delegate void BallLaunched(float flightPower, float velocity, Vector3
initial, Vector3 target);
    public BallLaunched OnBallLaunched;

    public bool HasBall { get; set; }

    public float BallFlightTime { get; set; }

    public Vector3 BallHitTarget { get; set; }

    public Vector3 BallInitialPosition { get; internal set; }

    public GoalKeeperFSM FSM { get; set; }

    public RPGMovement RPGMovement { get; set; }

    private void Awake()
    {
       FSM = GetComponent<GoalKeeperFSM>();
```

```
        RPGMovement = GetComponent<RPGMovement>();
    }

    public bool IsBallWithChasingDistance()
    {
        return DistanceOfBallToGoal() <= 20f;
    }

    public bool IsBallWithThreateningDistance()
    {
        return DistanceOfBallToGoal() <= 30f;
    }

    public bool IsShotOnTarget()
    {
        return _goal.IsPositionWithinGoalMouthFrustrum(BallHitTarget);
    }

    public float DistanceOfBallToGoal()
    {
        return Vector3.Distance(_ball.transform.position, _goal.transform.position);
    }

    public void Instance_OnBallLaunched(float flightTime, float velocity, Vector3
initial, Vector3 target)
    {
        BallLaunched temp = OnBallLaunched;
```

```
    if (temp != null)
        temp.Invoke(flightTime, velocity, initial, target);
}

public Vector3 Position
{
    get
    {
        return transform.position;
    }
}

public float DiveReach
{
    get
    {
        return JumpDistance + Reach;
    }
}

public float DiveSpeed
{
    get
    {
        return _diveSpeed;
    }
```

```
    set
    {
        _diveSpeed = value;
    }
}

public float GoalKeeping
{
    get
    {
        return _goalKeeping;
    }

    set
    {
        _goalKeeping = value;
    }
}

public float JumpDistance
{
    get
    {
        return _jumpDistance;
    }

    set
```

```
    {
        _jumpDistance = value;
    }
}

public float JumpReach
{
    get
    {
        return Height + JumpHeight;
    }
}

public float Reach
{
    get
    {
        return _reach;
    }

    set
    {
        _reach = value;
    }
}

public float TendGoalDistance
```

```
{
    get
    {
        return _tendGoalDistance;
    }

    set
    {
        _tendGoalDistance = value;
    }
}

public float TendGoalSpeed
{
    get
    {
        return _tendGoalSpeed;
    }

    set
    {
        _tendGoalSpeed = value;
    }
}

public Animator Animator
{
```

```
    get
    {
        return _animator;
    }

    set
    {
        _animator = value;
    }
}

public Ball Ball
{
    get
    {
        return _ball;
    }

    set
    {
        _ball = value;
    }
}

public Goal Goal
{
    get
```

```
    {
        return _goal;
    }

    set
    {
        _goal = value;
    }
}

public float Height
{
    get
    {
        return _height;
    }

    set
    {
        _height = value;
    }
}

public float JumpHeight
{
    get
    {
```

```
            return _jumpHeight;
        }

        set
        {
            _jumpHeight = value;
        }
    }

    public Transform ModelRoot
    {
        get
        {
            return _modelRoot;
        }

        set
        {
            _modelRoot = value;
        }
    }

    public float BallVelocity { get; internal set; }
  }
}
```

5.9.InterceptShotMainState.cs:

```
using Assets.SuperGoalie.Scripts.Entities;
using Assets.SuperGoalie.Scripts.FSMs;
using Assets.SuperGoalie.Scripts.Others.Utilities;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Dive.MainState;
using RobustFSM.Base;
using UnityEngine;

namespace
Assets.SuperGoalie.Scripts.States.GoalKeeperStates.InterceptShot.MainState
{
    public class InterceptShotMainState : BState
    {
        float _height;
        float _initDistOfBallToOrthogonalPoint;
        float _speed;
        float _timeToOrthogonalPoint;
        float _timeOfBallToPlayerOrthogonalPointOnBallPath;
        float _timeOfBallToPlayerOrthogonalPointOnBallPathCached;
        float _turn;
        float _weightMultiplier;
        Vector3 _ballInitPosition;
        Vector3 _playerInterceptPoint;
        Vector3 _playerSteeringTarget;
        Vector3 _ballPositionAtPlayerOrthogonalPoint;
        Vector3 _ballVelocity;
```

```
    Vector3 _ballVelocityXZ;
    Vector3 _normalizedBallInitialPosition;
    Vector3 _normalizedBallPosition;
    Vector3 _normalizedBallTarget;
    Vector3 _normalizedPlayerPosition;
    Vector3 _relativeBallPositionAtPlayerInterceptPoint;
    Vector3 _orthogonalPointToPlayerPositionOnBallPath;

    public bool BallTrapable { get; set; }

    public float RequiredJumpHeight { get; set; }

    public Vector3 LeftHandTargetPosition { get; set; }

    public Vector3 RightHandTargetPosition { get; set; }

    public override void Enter()
    {
        base.Enter();

        //normalize stuff
        _normalizedBallInitialPosition = new Vector3(Owner.BallInitialPosition.x, 0f,
Owner.BallInitialPosition.z);
        _normalizedBallTarget = new Vector3(Owner.BallHitTarget.x, 0f,
Owner.BallHitTarget.z);
        _normalizedPlayerPosition = new Vector3(Owner.Position.x, 0f,
Owner.Position.z);
```

```
        //find the point on the ball path to target that is orthogonal to player position

        _playerInterceptPoint = _orthogonalPointToPlayerPositionOnBallPath =
OrthogonalPoint.OrthPoint(_normalizedBallInitialPosition, _normalizedBallTarget,
Owner.Position);


        //calculate some data depended on the orthogonal point

        _initDistOfBallToOrthogonalPoint =
Vector3.Distance(_orthogonalPointToPlayerPositionOnBallPath,
_normalizedBallInitialPosition);

        float distanceBetweenInitialBallPositionAndOrthogonalPoint =
Vector3.Distance(_normalizedBallInitialPosition,
_orthogonalPointToPlayerPositionOnBallPath);

        _timeOfBallToPlayerOrthogonalPointOnBallPathCached =
_timeOfBallToPlayerOrthogonalPointOnBallPath =
distanceBetweenInitialBallPositionAndOrthogonalPoint / Owner.BallVelocity;


        //calculate ball position at player orthogonal point

        _ballPositionAtPlayerOrthogonalPoint =
Owner.Ball.FuturePosition(_timeOfBallToPlayerOrthogonalPointOnBallPath);


        /**Calculate player control variables**/


        Vector3 dirToOrthogonalPoint =
_orthogonalPointToPlayerPositionOnBallPath - Owner.Position;
//calculate the player intercept point

        dirToOrthogonalPoint.y = 0.0f;


        float maxMovementDistanceOfPlayer = 0f;

        float playerDistanceToOrthogonalPoint = dirToOrthogonalPoint.magnitude;


        //find the distance that the player can jump and still reach the ball
```

```
        if (playerDistanceToOrthogonalPoint >= Owner.Reach)
            maxMovementDistanceOfPlayer =
Mathf.Clamp(playerDistanceToOrthogonalPoint - Owner.Reach, 0f,
Owner.JumpDistance);        //find the maximum distance the player can dive
        else
            maxMovementDistanceOfPlayer = playerDistanceToOrthogonalPoint;


        //get the relative ball position at player
        _relativeBallPositionAtPlayerInterceptPoint =
Owner.transform.InverseTransformPoint(_ballPositionAtPlayerOrthogonalPoint);


        //if I'm already at target, I target somewhere in front of me
        if (Mathf.Abs(_relativeBallPositionAtPlayerInterceptPoint.x) < Owner.Reach)
        {
            //update the ball intercept point
            _playerInterceptPoint += Owner.transform.forward * Owner.Reach * 0.5f;
            _initDistOfBallToOrthogonalPoint =
Vector3.Distance(_playerInterceptPoint, _normalizedBallInitialPosition);


            //we get the future ball target at steering target
            _timeOfBallToPlayerOrthogonalPointOnBallPathCached =
_timeOfBallToPlayerOrthogonalPointOnBallPath =
_initDistOfBallToOrthogonalPoint / Owner.BallVelocity;
            _ballPositionAtPlayerOrthogonalPoint =
Owner.Ball.FuturePosition(_timeOfBallToPlayerOrthogonalPointOnBallPath);


            //find the future position of the ball relative to me
            _relativeBallPositionAtPlayerInterceptPoint =
Owner.transform.InverseTransformPoint(_ballPositionAtPlayerOrthogonalPoint);
        }
```

```
        //calculate the required player speed to reach the steering target at the same time as the ball

        float playerRawDiveSpeed = maxMovementDistanceOfPlayer / _timeOfBallToPlayerOrthogonalPointOnBallPath;          //find the raw dive speed

        float playerDiveSpeed = Mathf.Clamp(playerRawDiveSpeed, 0f, Owner.DiveSpeed);                    //clamp the player dive speed


        //calculate the jump height required to reach the ball

        float playerRawJumpHeight = _relativeBallPositionAtPlayerInterceptPoint.y - Owner.Height;                    //find the jump height to reach ball

        float playerJumpHeight = Mathf.Clamp(playerRawJumpHeight, 0f, Owner.JumpHeight);                    //clamp the jump height

        RequiredJumpHeight = playerJumpHeight;


        //find the player intecpt point

        _playerSteeringTarget = _normalizedPlayerPosition + dirToOrthogonalPoint.normalized * maxMovementDistanceOfPlayer;


        //set the steering component

        Owner.RPGMovement.SetMoveTarget(_playerSteeringTarget);

        Owner.RPGMovement.Speed = playerDiveSpeed;

        Owner.RPGMovement.SetSteeringOn();


        //set the height animator value

        _height = Mathf.Clamp(_relativeBallPositionAtPlayerInterceptPoint.y / Owner.JumpReach, 0f, 1f);


        //set the turn animator value
```

```
        if (Mathf.Abs(_relativeBallPositionAtPlayerInterceptPoint.x) < Owner.Reach)
            _turn = 0f;
        else if (_relativeBallPositionAtPlayerInterceptPoint.x > 0)
            _turn = 1f;
        else if (_relativeBallPositionAtPlayerInterceptPoint.x < 0)
            _turn = -1f;

        //set the dive animation to play
        Owner.Animator.SetFloat("Height", _height);
        Owner.Animator.SetFloat("Turn", _turn);
        Owner.Animator.SetTrigger("Dive");

        //calculate the targets for the hands
        LeftHandTargetPosition = GetBoneIKTarget(HumanBodyBones.LeftHand);
        RightHandTargetPosition =
GetBoneIKTarget(HumanBodyBones.RightHand);

    }

    public override void Execute()
    {
        base.Execute();

        //normalize stuff here
        _normalizedBallPosition = new Vector3(Owner.Ball.Position.x,
Owner.Ball.Position.y + Owner.Ball.SphereCollider.radius, Owner.Ball.Position.z);
        _normalizedPlayerPosition = new Vector3(Owner.Position.x, 0f,
Owner.Position.z);
```

```
        //stop moving if at target
        if (Vector3.Distance(_normalizedPlayerPosition, _playerSteeringTarget) <=
0.1f)
            Owner.RPGMovement.SetSteeringOff();

        //punch the ball
        if (_turn == 0f)
        {
            //check if the right hand can punch the ball
            BallTrapable = GetDistOfBoneToPosition(HumanBodyBones.RightHand,
_normalizedBallPosition) <=
GetDistanceTravelledInSingleFrame(Owner.Ball.Rigidbody.velocity)
                || GetDistOfBoneToPosition(HumanBodyBones.LeftHand,
_normalizedBallPosition) <=
GetDistanceTravelledInSingleFrame(Owner.Ball.Rigidbody.velocity);

            //punch the ball
            if (BallTrapable)
                Machine.ChangeState<PunchBallMainState>();
        }
        else if (_turn == 1)
        {
            //check if the right hand can punch the ball
            float distanceBetweenHandAndBall =
GetDistOfBoneToPosition(HumanBodyBones.RightHand, _normalizedBallPosition);
            BallTrapable = distanceBetweenHandAndBall <=
GetDistanceTravelledInSingleFrame(Owner.Ball.Rigidbody.velocity);

            //punch the ball
```

```
            if (BallTrapable)
                Machine.ChangeState<PunchBallMainState>();
        }
        else if (_turn == -1)
        {
            //check if the left hand can punch the ball
            float distanceBetweenHandAndBall =
GetDistOfBoneToPosition(HumanBodyBones.LeftHand, _normalizedBallPosition);
            BallTrapable = distanceBetweenHandAndBall <=
GetDistanceTravelledInSingleFrame(Owner.Ball.Rigidbody.velocity);

            //punch the ball
            if (BallTrapable)
                Machine.ChangeState<PunchBallMainState>();
        }

        //check if I have exhausted my dive time
        //exit if dive time exhausted
        _timeOfBallToPlayerOrthogonalPointOnBallPath -= Time.deltaTime;
        if (_timeOfBallToPlayerOrthogonalPointOnBallPath <= 0f)
            Machine.ChangeState<PunchBallMainState>();
    }

    public override void Exit()
    {
        base.Exit();

        //set the steering to off
```

```
        Owner.RPGMovement.SetSteeringOff();

        //set the animator to exit the dive state
        Owner.Animator.ResetTrigger("Dive");
    }

    public override void OnAnimatorIK(int layerIndex)
    {
        base.OnAnimatorIK(layerIndex);

        //calculate the weight multiplier depending on the remaining distance to target
        _normalizedBallPosition = new Vector3(Owner.Ball.Position.x, 0f,
Owner.Ball.Position.z);
        _normalizedPlayerPosition = new Vector3(Owner.Position.x, 0f,
Owner.Position.z);

        //find the distance of ball to orthogonal point
        float distanceOfBallToTarget = Vector3.Distance(_normalizedBallPosition,
_orthogonalPointToPlayerPositionOnBallPath);

        //if ball comes within reach influence the weight multiplier
        if (distanceOfBallToTarget > 5 * Owner.Reach)
            _weightMultiplier = 0f;
        else
            _weightMultiplier = (5 * Owner.Reach - distanceOfBallToTarget) / 5 *
Owner.Reach;

        float leftHandWeight = 0f;
```

```
        float rightHandWeight = 0f;
        float lookAtWeight = 0f;

        //choose which hands to effect
        if (_turn == 0f)
        {
            leftHandWeight = _weightMultiplier;
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;
        }
        else if(_turn == -1)
        {
            leftHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;
        }
        else if(_turn == 1)
        {
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;
        }

        //set the animations weights
        Owner.Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand,
leftHandWeight);
        Owner.Animator.SetIKPositionWeight(AvatarIKGoal.RightHand,
rightHandWeight);
        Owner.Animator.SetLookAtWeight(lookAtWeight);
```

```
        //set the animations positions

        Owner.Animator.SetLookAtPosition(Owner.Ball.Position + new Vector3(0f,
Owner.Ball.SphereCollider.radius, 0f));

        Owner.Animator.SetIKPosition(AvatarIKGoal.LeftHand,
LeftHandTargetPosition);

        Owner.Animator.SetIKPosition(AvatarIKGoal.RightHand,
RightHandTargetPosition);

    }


    public override void OnAnimatorMove()

    {

        base.OnAnimatorMove();


        //calculate the ratio to ball point

        float ratio = (_timeOfBallToPlayerOrthogonalPointOnBallPathCached -
_timeOfBallToPlayerOrthogonalPointOnBallPath) /
_timeOfBallToPlayerOrthogonalPointOnBallPathCached;


        //manipulate the player height

        float positionY = Mathf.Lerp(0f, RequiredJumpHeight, ratio);


        //now move the player character depending on the height

        Vector3 localPosition = new
Vector3(Owner.ModelRoot.transform.localPosition.x, positionY,
Owner.ModelRoot.transform.localPosition.z);

        Owner.ModelRoot.transform.localPosition = localPosition;

    }


    public Transform GetBone(HumanBodyBones bone)

    {
```

```
        return Owner.Animator.GetBoneTransform(bone).transform;
    }

    public float GetDistOfBoneToPosition(HumanBodyBones bone, Vector3
position)
    {
        //find the distance between the bone and the target
        return
Vector3.Distance(Owner.Animator.GetBoneTransform(bone).transform.position,
position);
    }

    public float GetDistanceTravelledInSingleFrame(Vector3 velocity)
    {
        return velocity.magnitude * Time.deltaTime;
    }

    public Vector3 GetBoneIKTarget(HumanBodyBones bone)
    {
        //prepare data to calculate hit target
        Vector3 ballIKTarget = _ballPositionAtPlayerOrthogonalPoint + new
Vector3(0f, Owner.Ball.SphereCollider.radius, 0f);
        //Vector3 bonePosition = GetBone(bone).position;
        //Vector3 directionOfIkTargetToBone = bonePosition - ballIKTarget;

        //calculate the ik target
        //ballIKTarget = bonePosition + directionOfIkTargetToBone.normalized *
(directionOfIkTargetToBone.magnitude - Owner.Ball.SphereCollider.radius);
```

```
        //ballIKTarget = ballIKTarget + directionOfIkTargetToBone.normalized *
Owner.Ball.SphereCollider.radius;


        //return the ik target
        return ballIKTarget;
    }

    GoalKeeper Owner
    {
        get
        {
            return ((GoalKeeperFSM)SuperMachine).Owner;
        }
    }

    public float Turn
    {
        get
        {
            return _turn;
        }

        set
        {
            _turn = value;
        }
    }
```

```
    }
}
```

5.10.PunchBallMainState.cs:

```
using Assets.SuperGoalie.Scripts.Entities;
using Assets.SuperGoalie.Scripts.FSMs;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Idle.MainState;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.InterceptShot.MainState;
using RobustFSM.Base;
using System;
using UnityEngine;

namespace Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Dive.MainState
{
    public class PunchBallMainState : BState
    {
        bool _ballTrapable;
        float _height;
        float _time;
        float _turn;
        float _weightMultiplier;
        Vector3 _leftHandTargetPosition;
        Vector3 _rightHandTargetPosition;

        public override void Enter()
        {
            base.Enter();
```

```
        _time = 0f;


        //get some important data
        _ballTrapable = Machine.GetState<InterceptShotMainState>().BallTrapable;
        _leftHandTargetPosition =
Machine.GetState<InterceptShotMainState>().LeftHandTargetPosition;
        _rightHandTargetPosition =
Machine.GetState<InterceptShotMainState>().RightHandTargetPosition;
        _turn = Machine.GetState<InterceptShotMainState>().Turn;


        //if the ball is trappable then hit it away
        if (_ballTrapable)
        {
            //calculate the punch direction
            Vector3 ballRelativePosition =
Owner.transform.InverseTransformPoint(Owner.Ball.Position);
            Vector3 ballPunchDirection = Vector3.zero;


            //detemine the punch direction
            if (Mathf.Abs(ballRelativePosition.x) > 0.1f)
            {
                //simply punch it to the side
                ballPunchDirection = new Vector3(ballRelativePosition.x, 0f, 0f);
            }
            else
            {
                //if it's less than my height then punch it infront of me else punch up
```

```
            if (ballRelativePosition.y <= Owner.Height)
                ballPunchDirection = new Vector3(0f, 0f, 1f);
            else
                ballPunchDirection = new Vector3(0f, 1f, -1f);
        }

        //punch the ball
        ballPunchDirection =
Owner.transform.TransformDirection(ballPunchDirection);
        ballPunchDirection.Normalize();
        Owner.Ball.Rigidbody.velocity = ballPunchDirection * 0.5f *
Owner.Ball.Rigidbody.velocity.magnitude;
    }

    //set the animator to exit the dive state
    Owner.Animator.SetTrigger("Exit");

    //raise the punch ball event
    Action temp = Owner.OnPunchBall;
    if (temp != null)
        temp.Invoke();
}

public override void Execute()
{
    base.Execute();

    //go to idle state the moment the player gets into idle state
```

```
        if (Owner.Animator.GetCurrentAnimatorStateInfo(0).IsName("Idle"))
            Machine.ChangeState<IdleMainState>();
    }

    public override void OnAnimatorIK(int layerIndex)
    {
        base.OnAnimatorIK(layerIndex);

        //declare the weights
        float leftHandWeight = 0f;
        float rightHandWeight = 0f;
        float lookAtWeight = 0f;

        //set the time
        if(_time < 1f)
            _time += 10 * Time.deltaTime;

        //set the weight multiplier
        _weightMultiplier = Mathf.Lerp(1f, 0f, _time);

        //choose which hands to effect
        if (_turn == 0f)
        {
            //set the weights
            leftHandWeight = _weightMultiplier;
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;
```

```
            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand,
leftHandWeight);
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.RightHand,
rightHandWeight);


            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.LeftHand,
_leftHandTargetPosition);
            Owner.Animator.SetIKPosition(AvatarIKGoal.RightHand,
_rightHandTargetPosition);
        }
        else if (_turn == -1)
        {
            //set the weights
            leftHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;


            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand,
leftHandWeight);


            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.LeftHand,
_leftHandTargetPosition);


        }
        else if (_turn == 1)
```

```
        {
            //set the weights
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;

            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.RightHand,
rightHandWeight);

            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.RightHand,
_rightHandTargetPosition);
        }

        //set the look target
        Owner.Animator.SetLookAtWeight(lookAtWeight);
        Owner.Animator.SetLookAtPosition(Owner.Ball.Position);
    }

    public override void OnAnimatorMove()
    {
        base.OnAnimatorMove();

        //manipulate the player height
        Owner.ModelRoot.transform.localPosition = Vector3.zero;
    }

    GoalKeeper Owner
```

```
        {
            get
            {
                return ((GoalKeeperFSM)SuperMachine).Owner;
            }
        }
    }
}
```

5.11.TendGoalMainState:

```
using Assets.SuperGoalie.Scripts.Entities;
using Assets.SuperGoalie.Scripts.FSMs;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Idle.MainState;
using Assets.SuperGoalie.Scripts.States.GoalKeeperStates.InterceptShot.MainState;
using RobustFSM.Base;
using System;
using UnityEngine;

namespace Assets.SuperGoalie.Scripts.States.GoalKeeperStates.Dive.MainState
{
    public class PunchBallMainState : BState
    {
        bool _ballTrapable;
        float _height;
        float _time;
        float _turn;
        float _weightMultiplier;
```

```
    Vector3 _leftHandTargetPosition;
    Vector3 _rightHandTargetPosition;

    public override void Enter()
    {
        base.Enter();

        _time = 0f;

        //get some important data
        _ballTrapable = Machine.GetState<InterceptShotMainState>().BallTrapable;
        _leftHandTargetPosition =
Machine.GetState<InterceptShotMainState>().LeftHandTargetPosition;
        _rightHandTargetPosition =
Machine.GetState<InterceptShotMainState>().RightHandTargetPosition;
        _turn = Machine.GetState<InterceptShotMainState>().Turn;

        //if the ball is trappable then hit it away
        if (_ballTrapable)
        {
            //calculate the punch direction
            Vector3 ballRelativePosition =
Owner.transform.InverseTransformPoint(Owner.Ball.Position);
            Vector3 ballPunchDirection = Vector3.zero;

            //detemine the punch direction
            if (Mathf.Abs(ballRelativePosition.x) > 0.1f)
            {
```

```
            //simply punch it to the side
            ballPunchDirection = new Vector3(ballRelativePosition.x, 0f, 0f);
        }
        else
        {
            //if it's less than my height then punch it infront of me else punch up
            if (ballRelativePosition.y <= Owner.Height)
                ballPunchDirection = new Vector3(0f, 0f, 1f);
            else
                ballPunchDirection = new Vector3(0f, 1f, -1f);
        }

        //punch the ball
        ballPunchDirection =
Owner.transform.TransformDirection(ballPunchDirection);
        ballPunchDirection.Normalize();
        Owner.Ball.Rigidbody.velocity = ballPunchDirection * 0.5f *
Owner.Ball.Rigidbody.velocity.magnitude;
    }

    //set the animator to exit the dive state
    Owner.Animator.SetTrigger("Exit");

    //raise the punch ball event
    Action temp = Owner.OnPunchBall;
    if (temp != null)
        temp.Invoke();
}
```

```
public override void Execute()
{
    base.Execute();

    //go to idle state the moment the player gets into idle state
    if (Owner.Animator.GetCurrentAnimatorStateInfo(0).IsName("Idle"))
        Machine.ChangeState<IdleMainState>();
}

public override void OnAnimatorIK(int layerIndex)
{
    base.OnAnimatorIK(layerIndex);

    //declare the weights
    float leftHandWeight = 0f;
    float rightHandWeight = 0f;
    float lookAtWeight = 0f;

    //set the time
    if(_time < 1f)
        _time += 10 * Time.deltaTime;

    //set the weight multiplier
    _weightMultiplier = Mathf.Lerp(1f, 0f, _time);

    //choose which hands to effect
```

```
        if (_turn == 0f)
        {
            //set the weights
            leftHandWeight = _weightMultiplier;
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;

            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand,
leftHandWeight);
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.RightHand,
rightHandWeight);

            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.LeftHand,
_leftHandTargetPosition);
            Owner.Animator.SetIKPosition(AvatarIKGoal.RightHand,
_rightHandTargetPosition);
        }
        else if (_turn == -1)
        {
            //set the weights
            leftHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;

            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand,
leftHandWeight);
```

```
            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.LeftHand,
_leftHandTargetPosition);

        }
        else if (_turn == 1)
        {
            //set the weights
            rightHandWeight = _weightMultiplier;
            lookAtWeight = _weightMultiplier;

            //set the animations weights
            Owner.Animator.SetIKPositionWeight(AvatarIKGoal.RightHand,
rightHandWeight);

            //set the animations positions
            Owner.Animator.SetIKPosition(AvatarIKGoal.RightHand,
_rightHandTargetPosition);
        }

        //set the look target
        Owner.Animator.SetLookAtWeight(lookAtWeight);
        Owner.Animator.SetLookAtPosition(Owner.Ball.Position);
    }

    public override void OnAnimatorMove()
    {
        base.OnAnimatorMove();
```

```
            //manipulate the player height
            Owner.ModelRoot.transform.localPosition = Vector3.zero;
        }

        GoalKeeper Owner
        {
            get
            {
                return ((GoalKeeperFSM)SuperMachine).Owner;
            }
        }
    }
}
```

5.12. pauseMenu.cs:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class pauseMenu : MonoBehaviour
{
    public GameObject pauseMenuUI;  // Reference to Pause Menu Panel
```

```
    public Button pauseButton;      // Pause Button
    public Button resumeButton;     // Resume Button
    public Button musicToggleButton; // Music Toggle Button
    public Button levelsButton;     // Go Back to Levels Button
    public AudioSource gameMusic;   // Background Music

    private bool isPaused = false;
    private bool isMusicOn = true;

    void Start()
    {
        // Ensure the pause menu is hidden at the start
        pauseMenuUI.SetActive(false);

        // Attach button listeners
        if (pauseButton) pauseButton.onClick.AddListener(TogglePause);
        if (resumeButton) resumeButton.onClick.AddListener(ResumeGame);
        if (musicToggleButton) musicToggleButton.onClick.AddListener(ToggleMusic);
        if (levelsButton) levelsButton.onClick.AddListener(GoBackToLevels);
    }

    public void TogglePause()
    {
        isPaused = !isPaused;
        pauseMenuUI.SetActive(isPaused);
        pauseButton.gameObject.SetActive(!isPaused); // Hide Pause Button when menu
is open
```

```
    Time.timeScale = isPaused ? 0 : 1;
}

public void ResumeGame()
{
    isPaused = false;
    pauseMenuUI.SetActive(false);
    pauseButton.gameObject.SetActive(true); // Show Pause Button again
    Time.timeScale = 1;
}

public void ToggleMusic()
{
    isMusicOn = !isMusicOn;
    if (gameMusic) gameMusic.mute = !isMusicOn;

    Text buttonText = musicToggleButton.GetComponentInChildren<Text>();
    if (buttonText)
        buttonText.text = isMusicOn ? "Turn Off Music" : "Turn On Music";
}

public void GoBackToLevels()
{
    Time.timeScale = 1;
    SceneManager.LoadScene("levels");
}
```

```
}
```

5.13.freekicklev.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class freekicklev : MonoBehaviour
{
  public Text playerDisplay;

  void Start() // <-- Change start() to Start()
  {
    if(DBManager.LoggedIn)
    {
       playerDisplay.text = "Player: " + DBManager.username;
    }
  }

  public void goToLevel1()
  {
     SceneManager.LoadScene("penald2");
  }

  public void goToLevel2()
```

```
{
    SceneManager.LoadScene("penald3");
}

public void goToLevel3()
{
    SceneManager.LoadScene("penald3");
}



public void goToLevel4()
{
    SceneManager.LoadScene("penald5");
}

 public void goToLevel5()
{
    SceneManager.LoadScene("penald6");
}

  public void goToLevel6()
{
    SceneManager.LoadScene("penald7");
}

 public void goToLevel7()
{
```

```
        SceneManager.LoadScene("penald8");
    }

     public void goToLevel8()
    {
        SceneManager.LoadScene("penald9");
    }

       public void goToLevel9()
    {
        SceneManager.LoadScene("penald10");
    }
    public void goToback()
    {
        SceneManager.LoadScene("levels");
    }
}
```

6.Results

6.1 Validation and Naming Conventions:

Sr. No	Control Id	Validation Used	Reason
1	username	Required Attribute	username of the User Cannot be Empty.
2	password	Required Attribute	Password Cannot be Empty.
3	confirmpassword	Required Attribute	Confirm Password Cannot be Empty.

1. Username

- Required Attribute: The username field is mandatory and cannot be left empty.
- Minimum Length Requirement: The username must contain at least 8 characters to ensure uniqueness and security.
- Alphanumeric Restriction: The username should consist of letters and numbers only, without special characters, to prevent potential SQL injection attacks.
- Uniqueness Constraint: The system must verify that the username does not already exist in the database before allowing the user to register.
- Case Sensitivity: The username should be case-insensitive, meaning "Player123" and "player123" should be treated as the same user.

2. Password

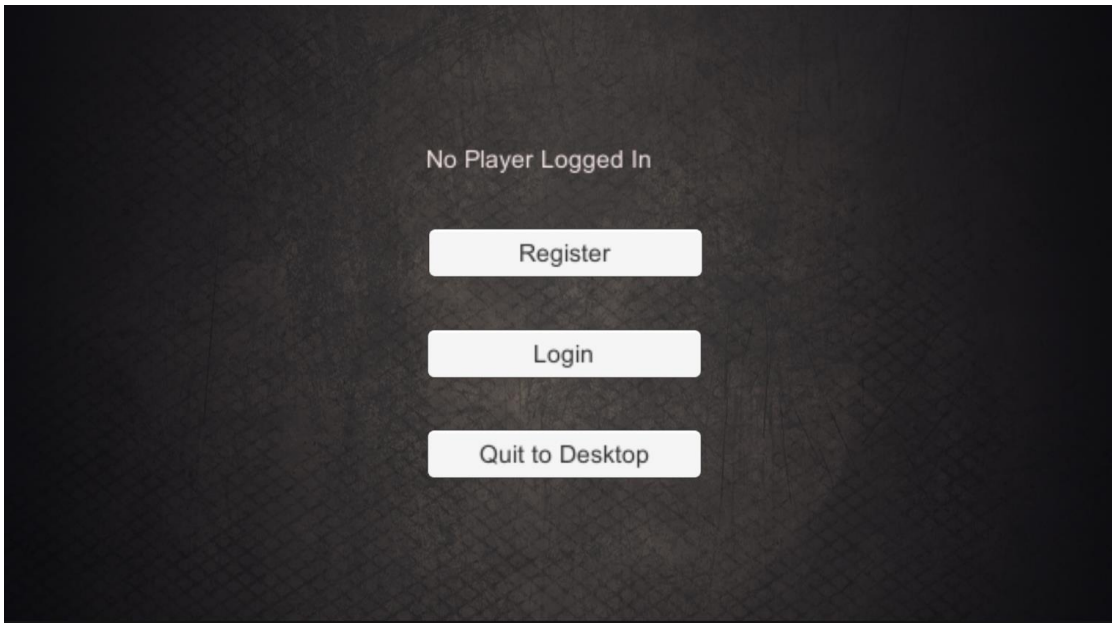
- Required Attribute: The password field must be filled out and cannot be left blank.
- Minimum Length: The password should be at least 8 characters long to enhance security.
- Complexity Requirement: The password must contain:
  - At least one uppercase letter (A-Z).
  - At least one lowercase letter (a-z).
  - At least one digit (0-9).
  - At least one special character (e.g., !, @, #, $).
- Hashing & Security: The password should be hashed before storage to prevent unauthorized access in case of database leaks.
- No Common Passwords: The system should prevent the use of common passwords (e.g., "password123", "12345678", "qwerty").

3. Confirm Password

- Required Attribute: This field is mandatory and cannot be left empty.
- Must Match Password: The value entered must exactly match the password field, ensuring that the user correctly enters their intended password.
- Real-time Validation: As the user types, an indicator should confirm whether the passwords match or not.
- Error Handling: If the passwords do not match, the system should display an error message prompting the user to re-enter them correctly.

6.2 Screenshots:

6.2.1 Registration scene:



The User Registration Screen is the entry point for players in the mobile football game. It is designed with a modern grey-themed background, offering a clean and minimalistic interface that prioritizes usability. The interface is structured to be intuitive, ensuring that both new and returning users can navigate the system effortlessly.

Screen Layout & Elements

Title Section:

- At the top of the screen, a bold white title displays the message: "No Player Logged In"

- This acts as an informative prompt, letting the user know they are currently not signed in and need to either register or log in to proceed.

Input Fields for User Credentials:

- Below the title, there are three input fields where users can enter their information.
- Each input field features light gray placeholder text, providing guidance on what details the user needs to enter.
- These fields are designed with a simple white background and slightly rounded edges, making them visually appealing and easy to interact with.

InteractiveButtons:
Below the input fields, there are three primary buttons, each serving a unique purpose:

- REGISTER Button:
  - This is a white button with bold black text labeled "REGISTER".
  - It allows new users to complete the registration process, creating a new account to access the game.
  - Upon clicking, the system verifies input details and securely stores the player's credentials.
- LOGIN Button:
  - This button, similar in style and size to the REGISTER button, is labeled "LOGIN".
  - It is designed for existing users who already have an account.
  - Upon selecting this option, users are redirected to the login screen, where they can enter their credentials to access their game profile.
- Quit to Desktop Button:
  - The Quit to Desktop button allows users to swiftly exit the application.
  - With a single click, players can terminate the game session and return to their device’s home screen.
  - This button ensures a quick and hassle-free exit, preventing unnecessary navigation through multiple menus.

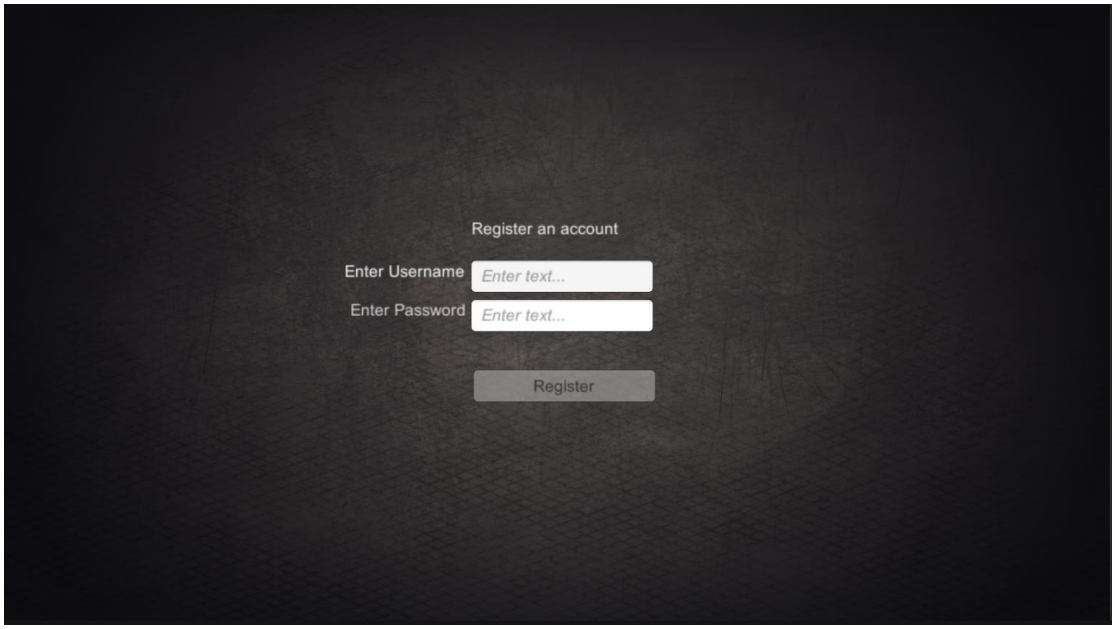
User Experience & Design Considerations

- The grey-themed background enhances readability and maintains a professional look.
- The contrast between white buttons and black text ensures visibility, making the interface accessible for users of all ages.
- Simple and minimal design prevents clutter, allowing for fast navigation and an easy registration/login process.
- The slightly rounded input fields and buttons contribute to a modern, mobile-friendly UI.

- The Quit to Desktop button ensures users can exit the game without frustration, offering a seamless and responsive user experience.

This User Registration Screen is designed to be efficient, user-friendly, and visually appealing, ensuring that new and returning players can quickly and securely access their game profiles

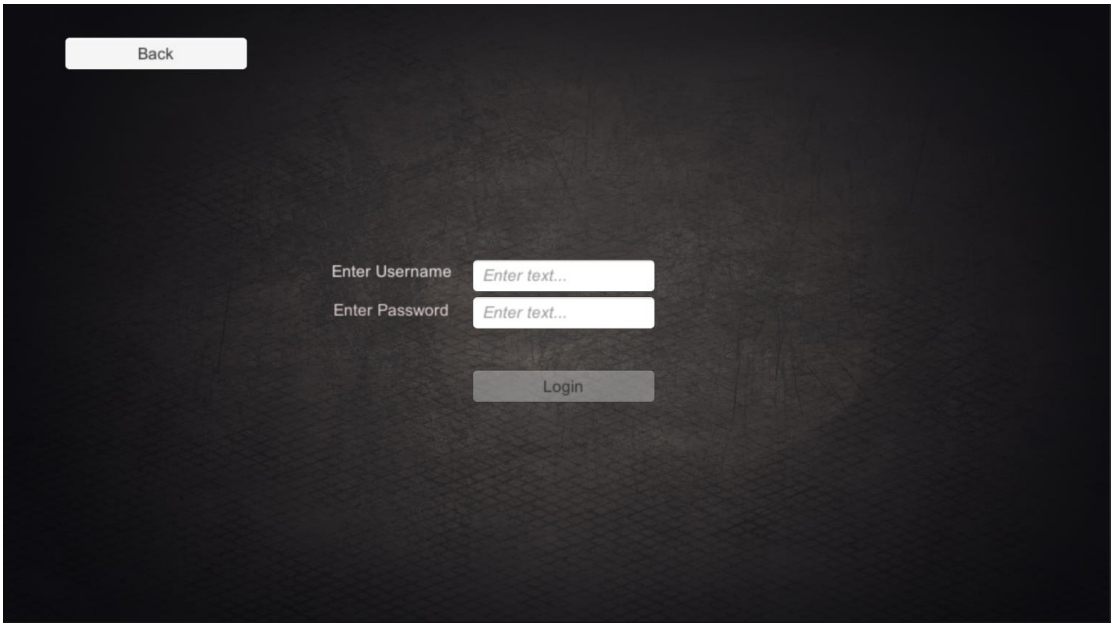
6.2.2 Register page:



Register Page:
The Register Page allows new users to create an account on the game application. Users provide essential details such as their name and password. There are validation checks to ensure strong passwords and unique usernames.
The page is designed with a simple and user-friendly form layout, making registration easy and fast. Once registered, users can immediately access:
Key Features:

- Contains input fields for username, password, and email.
- Implements validation checks to ensure that:
  - The username is unique and meets the character limit.
  - The password is strong enough (minimum characters required).
- A submit button triggers the registration process.
- If the registration is successful, the user is redirected back to LoginMenu to sign in.
- If registration fails (e.g., username already exists), the system simply comes to a halt.

6.2.3 Login Scene:



The LoginMenu scene serves as the entry point for returning users, ensuring that only registered and authenticated players can access the game's features. This scene plays a crucial role in user security and account management, allowing players to log in seamlessly with their existing credentials. The login system ensures that user data is validated before granting access, enhancing security and maintaining a smooth user experience.

Key Features of the LoginMenu Scene

Username and Password Input Fields

- The LoginMenu contains two primary input fields:
    - Username Field: Allows users to enter their registered username.
    - Password Field: A secure input field where players enter their password.
- The password field is designed to mask user input (displaying asterisks or dots) to prevent unauthorized access by onlookers.
- Placeholder text in each field provides guidance, ensuring users understand what information is required.

Submit Button – Authenticating User Credentials

- A Login/Submit button is present, which triggers the authentication process when clicked.
- Once clicked, the system verifies the provided credentials against the stored user data in the database.
- If the details match an existing user profile, the login is successful, and the player is redirected to the StartMenu scene, where they can proceed with gameplay.
- If authentication fails, the player is notified with a clear and informative error message.

Validation & Error Handling

- The system performs multiple validation checks before attempting to log in:
    - Empty Fields: If the username or password field is left blank, the system prompts the user to enter the required information before proceeding.

- Incorrect Credentials: If the entered username or password does not match an existing record, an error message is displayed, informing the user of incorrect details.
- Case Sensitivity: The login system ensures that username and password inputs match exactly as stored in the database, preventing login errors due to incorrect casing.

Redirect to RegistrationMenu – Account Creation

- If the user does not have an existing account, they have the option to navigate to the RegistrationMenu by clicking a "Sign Up" or "Create Account" button.
- This ensures that new players who have not yet registered are guided through the account creation process without confusion.

User Experience Enhancements

- The LoginMenu is designed to be user-friendly and visually intuitive, with:
  - A simple layout that minimizes distractions.
  - Clearly labeled fields and buttons to ensure easy navigation.
  - Quick response times for login attempts, ensuring a seamless user experience.
- The UI follows a minimalistic design, ensuring that players can log in with minimal effort and maximum efficiency.

Flow of User Interaction in the LoginMenu Scene

- The player enters their username and password into the respective fields.
- The player clicks the Submit/Login button.
- The system validates the input and checks the database for matching credentials.
- Successful login:
  - The player is redirected to the StartMenu scene, where they can select their preferred game mode.
- Unsuccessful login:
  - The system displays an error message, prompting the player to retry.
- If the player does not have an account, they can click "Sign Up", leading them to the RegistrationMenu for account creation.

Conclusion

The LoginMenu scene plays a vital role in the game's user authentication system, ensuring that only authorized players can access gameplay features. By implementing secure input handling, validation checks, and user-friendly navigation, the LoginMenu enhances the overall security and accessibility of the game. This scene serves as a gateway, allowing users to seamlessly log in, register, or recover their accounts, ensuring an efficient and hassle-free experience.

6.2.4 Start menu scene:

StartMenu Scene Overview

The StartMenu scene serves as the main hub of the game, providing players with access to different gameplay modes and essential navigation options. From this menu, players can choose to enter the Levels scene, where they can select different football challenges such as Penalty Shootout and Free Kick modes. Additionally, the menu provides an exit function that allows players to log out and return to the main game menu, ensuring a seamless transition between game states.

Key Features of the StartMenu Scene

Navigation Buttons for Seamless Movement

- The StartMenu contains multiple buttons that act as a central navigation hub, guiding players to various sections of the game.
- Each button is labeled clearly to ensure ease of use and a smooth user experience.

a) Levels Button – Accessing Different Game Modes

- Clicking this button takes the player to the Levels Scene, where they can choose from different game modes:

- Penalty Shootout Mode: A mode where players take penalty shots
- Freekick levels : A mode where players hit a freekick target
- Freekick Practice Mode: A mdoe where players can practice their freekick skills

6.2.5 levels scene:

The Levels Scene serves as a central hub where players can choose between various gameplay modes, offering a structured and intuitive selection interface. It provides easy access to both penalty shootout challenges and free kick levels, ensuring that players can seamlessly transition between different game experiences.

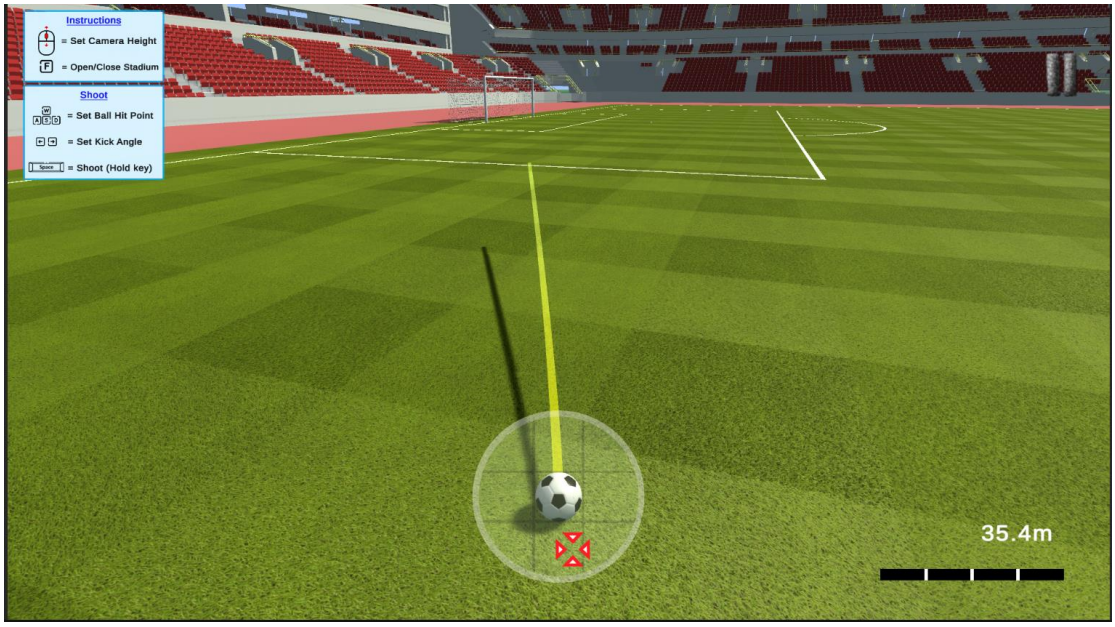
Key Features:

- Displays a visually engaging menu with clearly labeled buttons for each mode.
- Provides access to multiple game modes, catering to different playstyles:
  - Penalty Shootout Mode (pensocUL) – Unlimited penalty shootout gameplay.
  - Free Kick Challenges (freekicklev) – Levels focused on hitting target zones from set positions.
  - Various penalty challenge levels (penald1 - penald10) – Different structured penalty shootout challenges with increasing difficulty.
- Ensures smooth navigation between modes, allowing players to easily switch gameplay styles.
- Features return functionality so players can go back to the StartMenu if they want to explore other options.
- Offers an intuitive UI layout, ensuring players can quickly identify and select their preferred game mode.
- The scene contributes to a structured progression system, giving players the freedom to attempt different challenges at their own pace.

- Helps maintain a sense of player engagement and replayability by offering multiple pathways to experience the game.

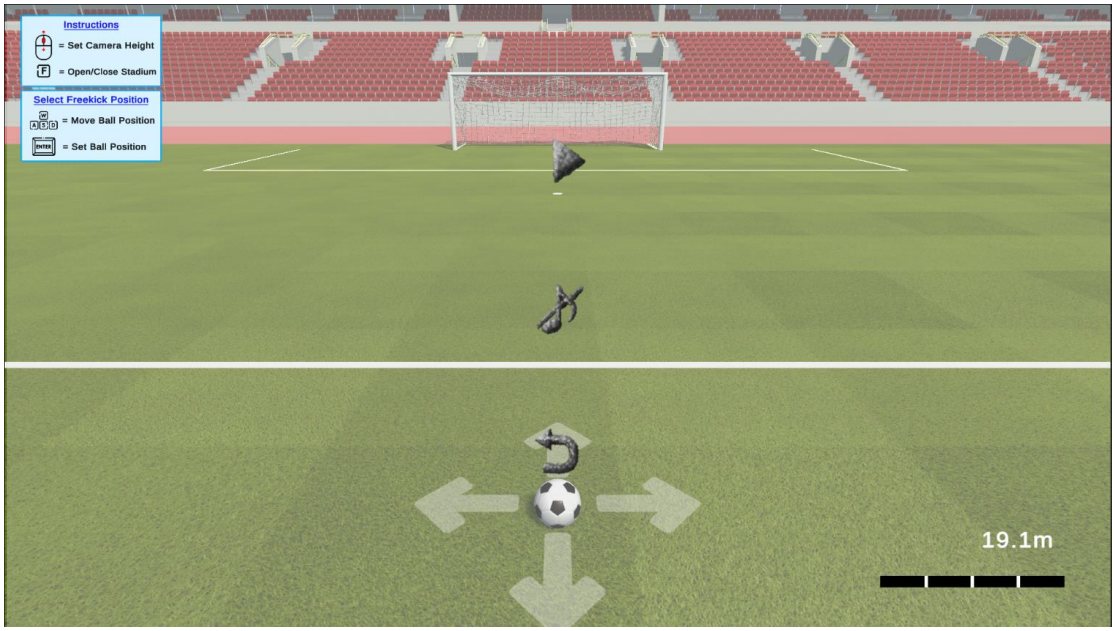
6.2.6 Freekick practice mode:

1.spawn environment, basic controls are explained by blue boxes on top left side with pause button on right side.



Ball in aiming mode after being moved from original position, you can notice how the distance from goal is show above the shot power meter

3.ball moved from original position after shooting



4. Pause menu in action with buttons like resume, music off and back button(heads back to levels scene)

Freekick Target Practice mode:

The Free Kick Target Practice Mode is a free-play scene where players can freely experiment with the game's shooting mechanics. Unlike other game modes that focus on scoring goals or hitting targets, this mode serves as a training ground where users can refine their ball control, shot power, and curve techniques without any restrictions or objectives.

Key Features:

- **Open Stadium Environment:** The player is placed inside a stadium with a football, goalposts, and a realistic field.

- **No Time Limit or Score:** Players can take unlimited shots without worrying about timers or scoring conditions.
- **Dynamic Ball Control:** Users can move the ball to different spots and adjust angles for varied practice shots.
- **Physics and Mechanics Testing:** Players can experiment with **curve shots, power shots, and lob shots** to understand ball movement and trajectory.
- **Background Music & Sound Effects:** Optional immersive stadium sounds and background music enhance the practice experience.
- **Pause Menu Access:** Players can pause the session and return to the main menu or levels menu at any time.

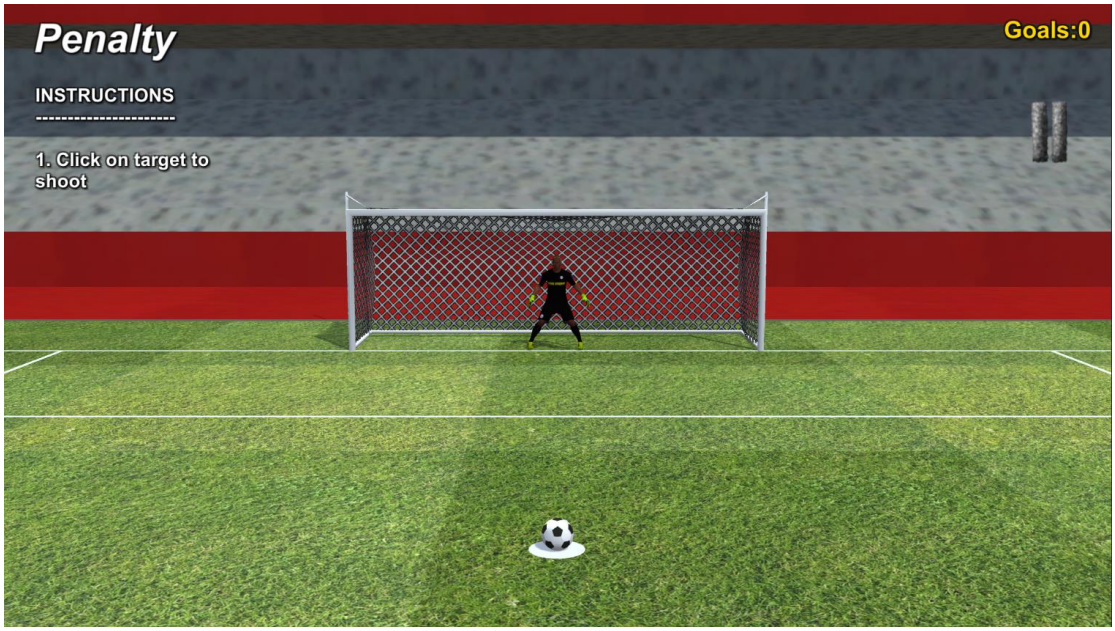
Gameplay Flow:

- The player enters the scene and can freely move the ball around.
- Shots can be taken repeatedly at the goalpost with no restrictions.
- Players can practice aiming, power control, and spin effects without pressure.
- When ready, players can use the pause menu to return to the main game modes.

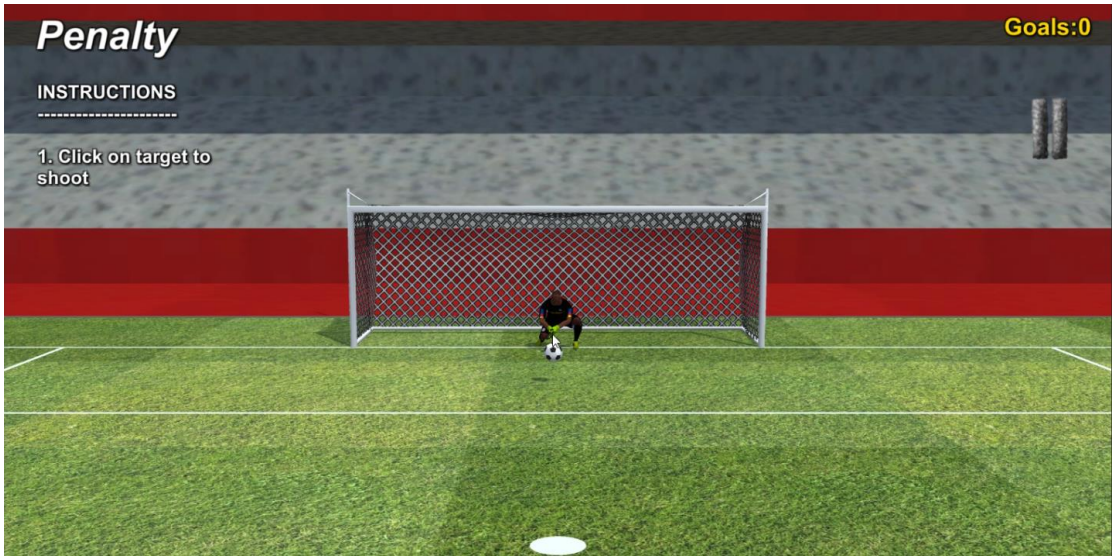
This scene is ideal for beginners who want to get comfortable with the game mechanics before attempting the more structured penalty shootout or free kick challenge levels.

6.2.7 Penalty-Shootout mode:

1.default spawn you can see the basic controls of aiming your mouse to shoot on left side where u must just simply aim mouse and shoot wherever you want to



2.Aiming at the goalkeeper and shooting

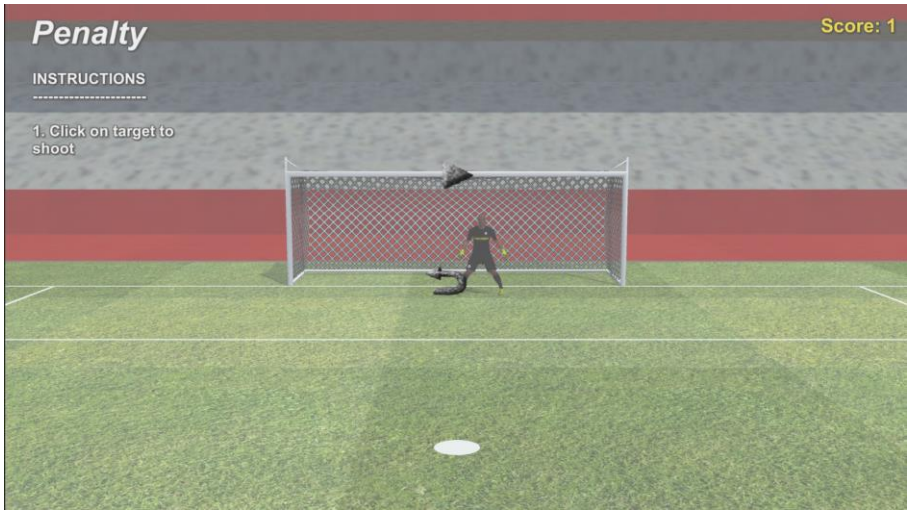


As you can see the ball moves in the direction we aimed at.

3.scoring goal

As you can see we scored a goal

4.pause menu



Unlimited Penalty Shootout Mode (pensocUL Scene)

The Unlimited Penalty Shootout Mode is designed to provide players with a continuous and immersive penalty kick experience, allowing them to take unlimited shots on goal. This game mode emphasizes skill, timing, and accuracy while gradually increasing the difficulty level, making it both engaging and challenging for the player.

This mode is ideal for practice, skill-building, and high-score chasing, as there is no shot limit, and the player can keep improving their shooting accuracy against an increasingly skilled goalkeeper. The more the player scores, the harder the challenge becomes, ensuring long-term engagement.

Key Features of the pensocUL Scene

Unlimited Shot Attempts

- Unlike standard penalty shootout modes that have a set number of attempts, this mode allows the player to take as many shots as they want.
- There is no end condition based on a number of shots, making this mode ideal for practice and skill mastery.

Immersive Crowd Atmosphere

- The crowd cheers the player on after every shot, creating a realistic stadium environment.
- Sound effects and animations enhance the sense of competition and excitement.

Dynamic Goalkeeper Difficulty

- As the player scores more goals, the goalkeeper's reactions and AI behavior become more difficult to beat.
- This progressive difficulty ensures that the mode remains challenging, requiring players to refine their skills over time.
- The goalkeeper anticipates shots better, reacts faster, and dives more accurately to block attempts.

Goal Counter and Score Feedback

- Every successful goal increments the goal counter, providing real-time feedback on the player's performance.
- This counter serves as an informal high-score tracker, encouraging players to set personal records.

Pause Menu for Game Control

- Players can access the Pause Menu at any time for in-game options.
- The Pause Menu provides three key functions:
    - Resume Game: Continues the unlimited penalty shootout mode.
    - Exit to Levels Scene: Ends the session and returns the player to the Levels selection screen.
    - Music Toggle Option: Players can turn background music on or off for a customizable experience.

Summary

The Unlimited Penalty Shootout Mode is an exciting, skill-based, and progressively challenging game mode that allows players to take an endless number of penalty shots without any predefined shot limit. This mode provides a continuous gameplay experience, allowing players to refine their shooting technique, improve their accuracy, and compete against an adaptive AI goalkeeper.
As the session progresses, the goalkeeper's difficulty increases, forcing players to develop better strategies and shot placements to score consistently. The real-time scoring feedback and dynamic crowd reactions create an immersive stadium-like

atmosphere, enhancing the realism and competitive spirit of the game. The unlimited nature of this mode makes it ideal for both casual play and serious training, allowing players to challenge themselves and aim for high scores.
The adaptive AI goalkeeper is a significant feature of this mode. Initially, the goalkeeper's reactions are slower and more predictable, making it easier for players to score. However, as the player continues to hit goals, the goalkeeper's response time, diving ability, and shot anticipation improve, making each shot progressively more difficult. This dynamic difficulty system ensures that the game remains engaging and prevents the experience from becoming too repetitive or easy.
The stadium ambiance and crowd effects further contribute to the excitement of this mode. The roaring cheers after each successful goal create a sense of accomplishment, while the tension builds as the goalkeeper becomes harder to beat. Visual elements such as stadium lighting, realistic player animations, and goal celebrations enhance the immersive football experience.

6.2.8 freekick-levels scene:

1.this scene shows the player all the available free kick levels present in the game

The Free Kick Level Scene serves as the central hub for structured free kick challenges, allowing players to select and attempt different levels designed to test their accuracy and precision in free kick situations. This scene presents players with a menu displaying nine distinct levels, each offering a unique setup with varying difficulties, obstacles, and target placements. The objective in each level is to successfully hit the target with a free kick, which will then trigger the completion of the level and allow the player to move forward.
Key Features:
Level Selection Menu

- The scene prominently features nine level buttons, each corresponding to a specific free kick challenge.

- Players can select any level to start their challenge, allowing non-linear progression through the game.
- The UI dynamically highlights completed levels, helping players track their progress.

Unique Free Kick Challenges

- Each level presents a different target that must be hit with a free kick to complete the challenge.
- As players advance, the difficulty increases, with more obstacles, varied target placements, and increased shot precision requirements.
- Some levels may include walls of defenders, moving targets, or tight goal angles to challenge players further.

Smooth UI and Navigation Options

- The user-friendly interface allows players to quickly browse and select levels.
- Players can return to the main menu or switch to penalty mode selection if they decide not to start a free kick challenge.
- Smooth transitions and animations enhance the overall user experience, making navigation intuitive and engaging.

Visual and Audio Feedback

- Upon selecting a level, players receive visual cues and audio feedback, confirming their choice.
- Successful completion of a level triggers a celebratory animation and sound effect, enhancing the sense of achievement.

Encouraging Player Progression

- Players are encouraged to improve their free kick accuracy by gradually attempting more difficult levels.
- The structured format provides clear objectives and a sense of progression, making it both engaging and rewarding.

Conclusion:

The Free Kick Level Scene acts as the primary hub for structured free kick gameplay, offering a clear progression system for players looking for goal-based challenges. By integrating engaging level selection mechanics, varied challenges, and seamless UI transitions, this scene enhances the game's replayability and provides a structured yet flexible approach to practicing free kicks. Whether players seek casual fun or competitive precision training, the Free Kick Level Scene ensures a compelling and challenging football experience.

6.2.9 Level 1:

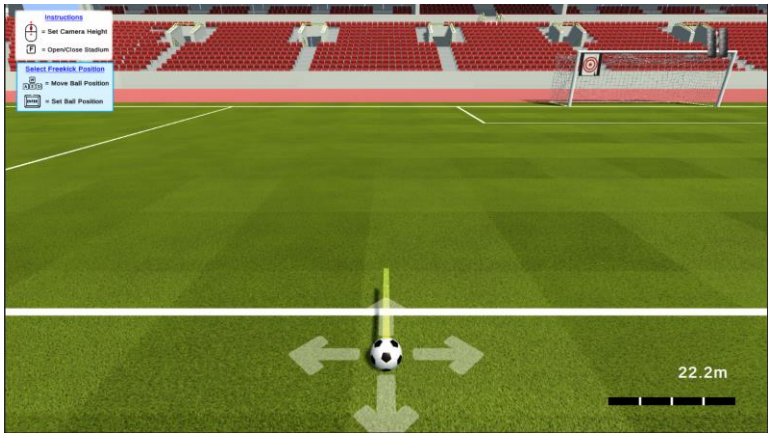
1.the ball spawns at the end of the box with target being on top left corner of goalkick

The player must use his wits to figure out the correct shooting position and hit the target to finish the level

The player must also note the ball is locked in place this time unlike the other levels

6.2.10 level 2:

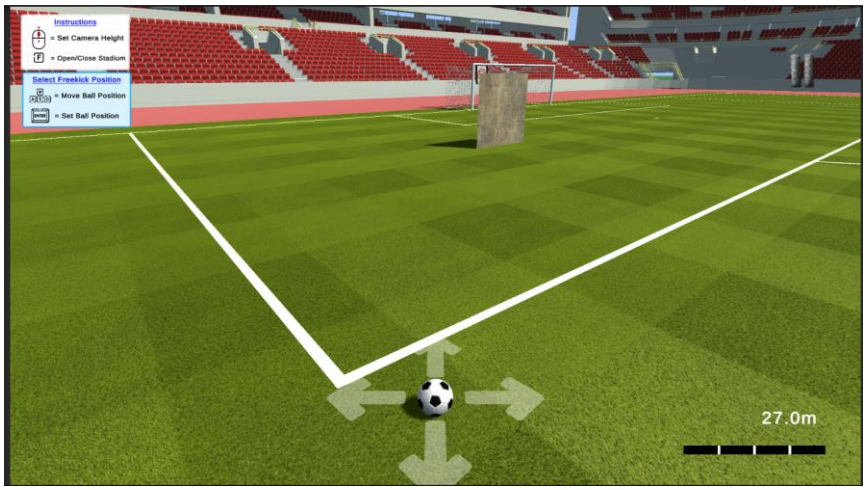
1.The ball spawns at the left side of the penalty box this time with the target still being in left upper side of the goal post



The player must manage to accurately hit the target to end the level.

6.2.11 level 3:

1.the ball is placed on the left corner of the box and this time there is a wall in between the target and the ball.

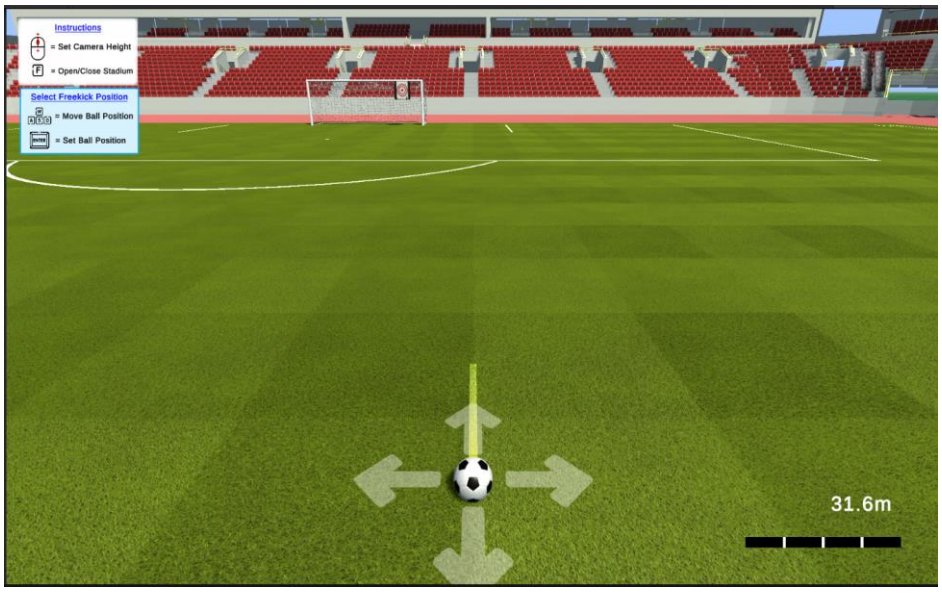


The player must accurately and precisely aim the ball over the wall in order to hit the target or else he will not be able to finish the level.

The player must aim higher with the ball in order to get the ball above the wall to the target.

6.2.12 level 4:

1.in this level the target is able to move and change its position



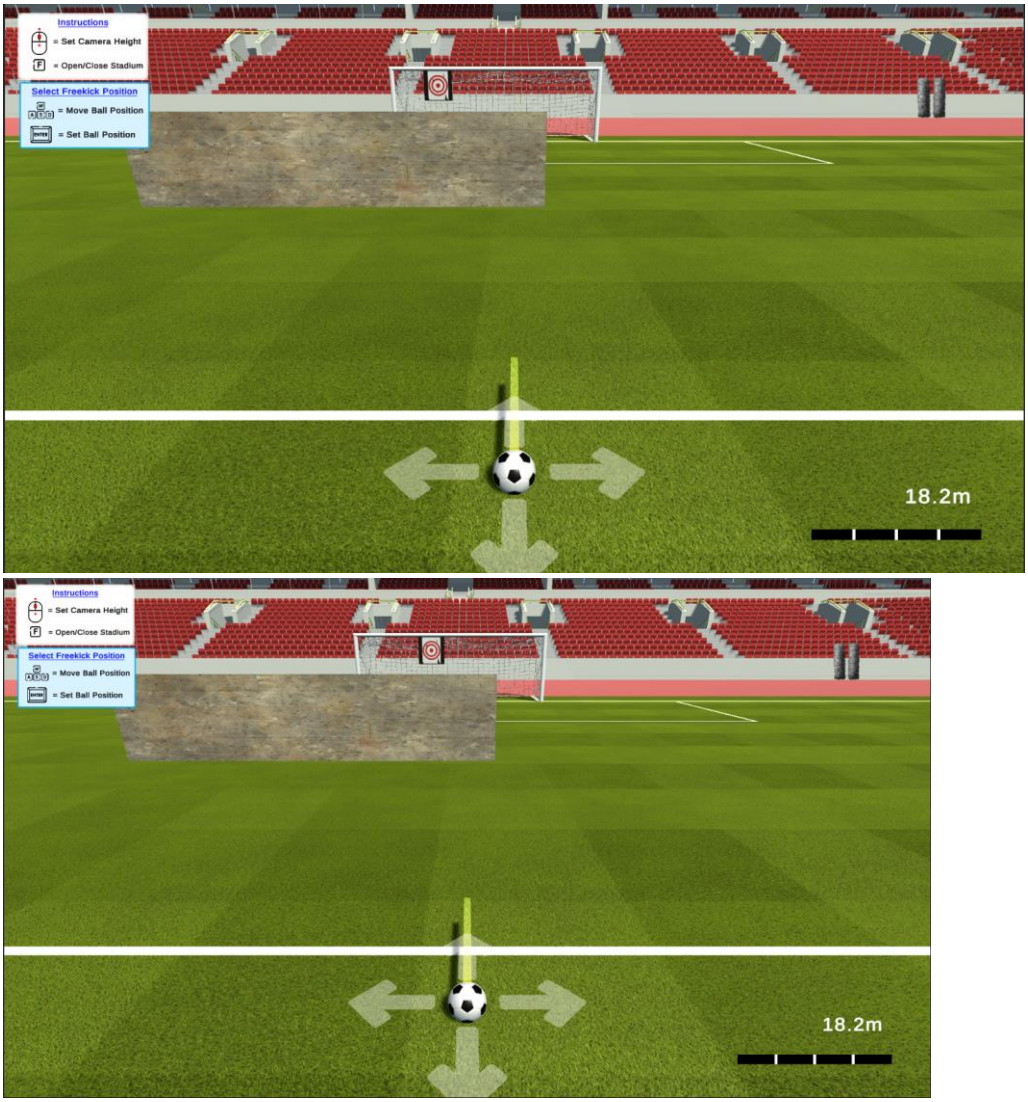
As we can see, the target in this level was successfully designed to move from left to right, adding an additional layer of challenge to the gameplay. Unlike static targets, a moving target requires greater precision, timing, and adaptability from the player.

To successfully hit the target, the player must not only accurately aim the ball but also carefully adjust their shot timing to compensate for the target's continuous movement. Since the target is positioned approximately 31 meters from the goal, this presents a considerable challenge, requiring the player to predict the movement trajectory and strike the ball at the perfect moment to ensure it reaches the intended spot.

The added movement factor encourages strategic thinking, as players must calculate both power and direction while also factoring in ball physics, such as curl and dip, to successfully land a shot on the moving target. Mastering this challenge will significantly improve the player's overall accuracy and reaction time, preparing them for even more difficult free kick scenarios in later levels.

6.2.13 level 5:

1.in this level the ball is closer to the target but the target is able to move left and right with a wall obstacle as well



We can see the ball moving
The player must accurately aim the ball over the wall to hit the target it shouldn't be hard as the goal distance is just 18 meters

6.2.14 level 6:

1.in this level the ball is spawned 23 meters from goal and the target moved up and down instead of sideways with a wall obstacle

In this level the player must note the 23 meters distance from goal accurately aim over the wall and judge the target movement in order to hit it accurately.

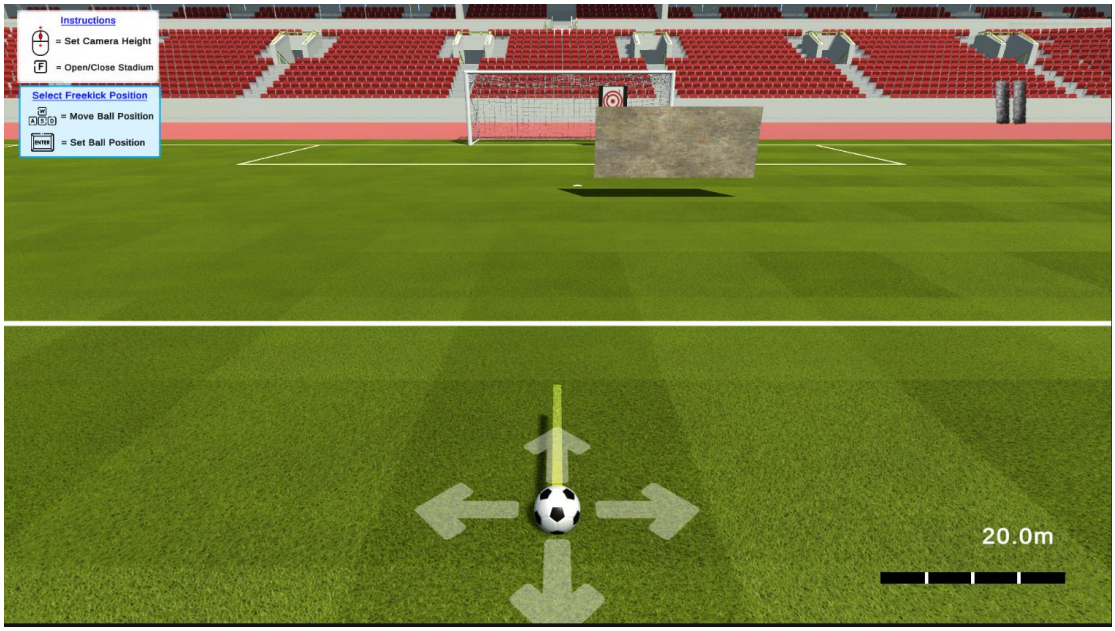
6.2.15 level 7:

1.the ball spawns in top right corner from the goal in this scene and in the target moves diagonally instead of left-right or up-down making it trickier to judge and there is a wall obstacle as well.

we can see the target move in diagonal directions
the player must accurately aim the ball over the wall from 23 meters to the goal and judge the movement of the target to hit it.

6.2.16 level 8:

1.in this level the ball spawns in front of the middle area of penalty box and both the wall and target move up-down in this situation,



As we can see both target and wall move together

In this level the player is given opening to hit the target in between the a=player must accurately judge it over the distance of 20 meters and hit the target accurately.

6.2.17 level 9:

1.in this level the ball spawns near middle area in front of the penalty box
This time two walls and target move left-right.

Being 23 meters away from goal the player must accurately judge the distance and precisely aim away from the wall to hit the target accurately.

7. Future Enhancement & conclusion:

7.1 Future Enhancement:

1. Multiplayer Mode
    - Implement online or local multiplayer so players can compete against friends.
    - Add leaderboards to track high scores and player rankings.
    - Introduce co-op gameplay, where two players can take turns in free kick challenges.
2. Advanced AI Opponents
    - Improve goalkeeper AI with adaptive difficulty based on player performance.
    - Add defensive walls that adjust dynamically to block shots in free kick levels.
    - Introduce AI-controlled defenders to simulate realistic match scenarios.
3. Customization Features
    - Enable ball and stadium customization

7.2 Conclusions:

This project successfully developed an engaging and interactive football game, designed to provide players with an immersive football experience that captures the excitement and challenges of real-world gameplay. By integrating precise shooting mechanics, dynamic ball physics, and responsive controls, the game creates a realistic simulation that encourages players to refine their skills, strategize their shots, and compete in various game modes.

One of the key strengths of this project is its intuitive gameplay mechanics, where shooting, aiming, and ball control are designed to feel natural and fluid. Players experience skill-based shooting, where factors such as angle, power, and ball placement influence the outcome of each shot. The inclusion of a free kick targeting system and penalty shootout challenges adds layers of depth, making every attempt feel unique and rewarding.

To enhance player engagement, the game implements smooth and responsive controls, ensuring an enjoyable playing experience across different platforms. The integration of a trajectory indicator for free kicks and penalties allows players to plan their shots strategically, while real-time animations and realistic goalkeeping AI add to the challenge and unpredictability of the game.

A significant feature of the project is the adaptive AI system, which ensures that goalkeepers react dynamically to player shots, creating a challenging yet fair experience. AI-driven opponents adjust their movements and save attempts based on player accuracy, making each session feel different and engaging. The game also includes structured levels and progressive challenges, motivating players to improve their shot precision and compete in increasingly difficult scenarios.

Beyond its current capabilities, the project establishes a strong foundation for future enhancements, ensuring long-term engagement and scalability. The modular design of the game allows for the seamless addition of advanced features, such as:

- multiplayer mode, enabling real-time competitive gameplay where players can challenge friends or participate in online penalty shootout duels

- career progression system, allowing players to advance through different levels, unlock new stadiums, and achieve milestones as they improve

- enhanced AI and difficulty customization, introducing smarter goalkeepers and varied challenge settings for personalized experiences

- expanded gameplay modes, including additional free kick challenges, trick shot modes, and dynamic weather effects to create varied match scenarios

In conclusion, this football game project successfully delivers an exciting and skill-based football experience, combining strategic gameplay, AI-driven challenges, and realistic mechanics. The smooth integration of shooting mechanics, goalkeeping AI, and level progression creates an immersive simulation that appeals to both casual players and competitive gamers alike. By offering multiple game modes, such as unlimited penalty shootouts and structured free kick challenges, this project provides a well-rounded football experience that encourages skill development and engagement.

Football video games have long been a staple in the gaming industry, offering players a digital experience that mirrors real-life soccer mechanics. Many games in this genre incorporate penalty shootouts, free kick challenges, and AI-controlled goalkeepers to create engaging and competitive gameplay. The implementation of physics-based ball movement, dynamic player interactions, and responsive goalkeeping AI enhances realism, making the gameplay more immersive and rewarding.

Furthermore, football games often feature user-friendly interfaces, pause menus**,** smooth scene transitions, and intuitive controls, ensuring that players can easily navigate between different game modes and settings. The ability to customize gameplay settings, practice in free play modes, and engage in structured challenges helps players refine their shooting techniques, decision-making skills, and overall game awareness.

With continuous advancements in game development, football simulation games continue to evolve, integrating multiplayer capabilities, career progression systems, and enhanced AI mechanics to provide players with a more dynamic and interactive experience. Future updates and enhancements could introduce online multiplayer matches, ranked leaderboards, team management features, and expanded game **modes** to further elevate the gaming experience.

Overall, this project successfully **captures** the essence of football by blending realistic mechanics, engaging challenges, and an intuitive interface. It provides players with an enjoyable and rewarding gameplay experience, while also laying the foundation for future improvements.

8.References:

For target movement
https://www.tutorialspoint.com/unity/unity_basic_movement_scripting.htm

for main menu
https://www.instructables.com/How-to-make-a-main-menu-in-Unity/
for ball physics
https://www.youtube.com/watch?v=ZOxnizAvMys
for database
https://learntocreategames.com/how-to-access-and-modify-a-database-from-unity/
for football goal physics
https://devforum.roblox.com/t/goaly-net-physics/2134126

9 Annexures

9.1 Figure list

4.1 Event Table: 7
4.2 Class Diagram: 8
4.3 Usecase Diagram: 10
4.4 Sequence Diagram: 12
4.5 Activity Diagram: 15
4.6. State Diagram 18
4.7 Package diagram: 19
4.8 Component diagram: 20
4.9 Deployment diagram: 22
4.10 Database Design: 23

9.2 Table list

4.10 Database Design: 23
6.1 Validation and Naming Conventions: 94