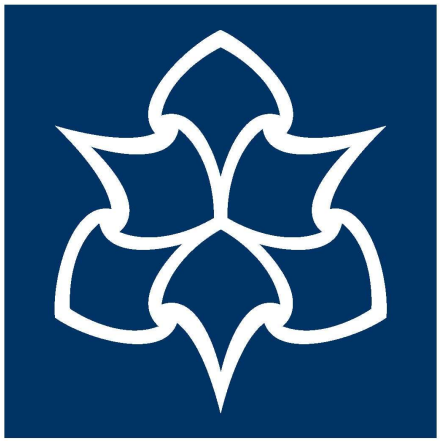
Ludo with Digital Dice

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF MASTER OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING



2024

By
Tawqeer Abdul Rahim Kapse
23735461
Department of Computing and Mathematics

Contents

Abstract

This dissertation explores the evolution of Ludo from its ancient origins as Pachisi to its modern-day digital transformation. Ludo, a game deeply rooted in Indian culture, has been adapted into several digital formats, particularly during the COVID-19 pandemic, when digital gaming became a key method of maintaining social connections. The study focuses on the comparative analysis of three Ludo platforms: the traditional Ludo board game, the mobile/PC version (Ludo King), and a hybrid version using digital dice with sensors.

The research employs a mixed-method approach, combining both qualitative and quantitative data from surveys and user reviews. The findings indicate that while the traditional Ludo board game offers a nostalgic and tactile experience, it is often perceived as time-consuming and effort-intensive. In contrast, the digital Ludo King is favored for its speed and convenience, making it the most preferred platform for casual gaming. The hybrid digital dice version strikes a balance between physical interaction and digital efficiency, appealing to those who appreciate a more immersive experience.

This study also delves into technological innovations in gaming hardware, such as digital dice and motion-sensing controllers, further expanding the scope of how traditional games are adapted to modern technology. The research highlights the importance of balancing nostalgia and technological advancement to enhance user experience in both digital and physical gaming formats.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number 68606.

Signed: Tawqeer Abdul Rahim Kapse

Date: 04/10/2024

1. Introduction

This paper presents a multifaceted exploration of traditional and digital gaming, with a particular focus on Ludo, and its adaptation from an ancient Indian game to a popular mobile application. The discussion is rooted in a historical overview, tracing Ludo's origins from the ancient game Pachisi and examining its transformation into a digital format, contextualizing this within the broader trend of traditional games being converted into mobile apps. This transformation reflects a significant shift in how games are experienced in the digital age, with Ludo serving as a prime example of a traditional board game that has successfully transitioned to digital platforms, especially during the COVID-19 pandemic.

The surge in Ludo's popularity during the pandemic is notable, as it highlights the game's role in maintaining social connections during lockdowns when physical interactions were limited. Ludo's digital adaptation provided a means for people to engage socially and pass time in a familiar yet technologically enhanced environment. The study delves into the functionality of nine popular Ludo apps, offering a detailed review of their features. This includes an examination of user interfaces, game modes, customization options, and unique functionalities such as auto-roll and token skins. The researchers observe that while the core gameplay mechanics remain consistent across different apps, the additional features vary, aiming to enhance user experience. The study's thematic analysis of 450 user reviews across these apps provides critical insights into player preferences and pain points, revealing a complex interplay between nostalgia, social interaction, and user satisfaction. Users appreciate the nostalgic value and social interaction facilitated by the digital versions of Ludo; however, they also express concerns over issues related to game fairness, in-app advertisements, and connectivity problems during online play. These findings culminate in recommendations for app developers, suggesting improvements in game mechanics, user interface design, and the handling of in-game advertisements to enhance overall player satisfaction.

In parallel, the exploration of gaming extends beyond Ludo to other implementations of digital and physical gaming systems, reflecting the broader intersection of technology,

entertainment, and education. This is exemplified in the paper "Implementation of a Digital Dice Game" by Jeena Joy and colleagues, which offers a comprehensive overview of the design and functionality of a digital dice game. This study underscores the application of digital electronics in game design, focusing on a microcontroller-based system that creates an engaging, interactive two-player game. The game employs a seven-segment display to simulate dice rolls and an LCD to track players' scores and turns. By setting a target score of 25, the game is designed to be straightforward yet competitive, demonstrating the successful integration of digital technology into traditional gaming formats. The project highlights several advantages of digital dice over traditional dice, such as the ability to generate any number from 0 to 9 and the enhanced visual feedback provided by the LCD. The authors also discuss potential modifications, such as increasing the number of players and altering the target score through slight programming adjustments, showcasing the flexibility and adaptability of digital gaming systems.

Further enriching this discourse, the paper "Real-time Implementation of Dice Unloading Algorithm" by Artem Vassilyev and colleagues from Tampere University of Technology presents an innovative approach to ensuring fairness in physical dice gameplay. The researchers address the inherent biases of physical dice through a novel algorithm that utilizes an inertial measurement unit (IMU) embedded in a foam-rubber die to capture real-time data. This data is processed to determine unbiased results, thus enhancing the fairness of tabletop games. The paper methodically outlines the problem of biased dice, the motivation for the study, and the development of the IMU-based die. It provides a detailed account of the hardware and software components, emphasizing the practical application of the proposed system. The results demonstrate the algorithm's effectiveness through various tests, illustrating the potential of this technology in creating more equitable gaming experiences.

Moreover, the relevance of gaming to educational contexts, particularly in making complex subjects like physics more engaging, is explored. Science education at the junior high school level, encompassing biology, physics, and chemistry, often sees students struggling with physics due to its perceived difficulty and abstract nature. Conventional teaching methods, which heavily rely on complex formulas, often fail to engage students effectively. To address this challenge, educators have experimented with various group learning methods to make physics more approachable and enjoyable, integrating game-based

learning as a potential solution. This approach aligns with the broader theme of the intersection of gaming and education, as seen in the digital adaptations of traditional games like Ludo and innovative projects like the digital dice game.

In addition to these developments, the current generation of gaming hardware has significantly expanded the possibilities for interactive and immersive experiences. Modern gaming interfaces, such as joysticks and controllers equipped with gyroscopes, microcontrollers, and sensors—exemplified by devices like PS Controllers, Wii Controllers, Xbox Controllers, and Nintendo Controllers—demonstrate the integration of sophisticated hardware in gaming. These devices provide users with more precise control and a more engaging gameplay experience by translating physical movements into in-game actions.

Moreover, advanced hardware such as the Tobii tracker, which uses eye-tracking technology, and the Leap Motion sensor, which captures hand and finger movements, have further pushed the boundaries of how players interact with games. These technologies enable more natural and intuitive interactions, enhancing the realism and immersion of gaming environments. Virtual reality (VR) and augmented reality (AR) headsets, such as the Oculus Rift, represent another leap forward, offering players fully immersive experiences where the boundaries between the game world and reality blur. These devices, coupled with powerful sensors and displays, create a gaming experience that is not only visually compelling but also physically engaging, as players can move and interact within a 3D space.

In conclusion, this study provides a comprehensive exploration of the diverse applications and impacts of gaming in both digital and physical formats. From the historical evolution of Ludo to the technological innovations in digital dice systems, the educational potential of game-based learning, and the advanced hardware that enhances interactivity and immersion, the research offers valuable insights into how gaming continues to evolve and influence various aspects of modern life. The findings underscore the importance of thoughtful design in enhancing user experience and ensuring fairness, whether in digital or physical gaming environments, and highlight the ongoing relevance of traditional games in the digital age. As gaming technology continues to advance, its role in entertainment, education, and social interaction is likely to expand, offering new opportunities and challenges for developers and players alike.

2. Literature Review

This literature survey provides an extensive review of the research related to Ludo, its variants, different types of dice, games that involve dice, and the current generation of gaming controllers that incorporate sensors and microcontrollers. This survey is structured to cover the historical evolution of Ludo, its digital transformation, the technical aspects of dice and their application in gaming, and the technological advancements in gaming hardware.

2.1. Historical Evolution of Ludo and Its Digital Transformation

Ludo, a game deeply rooted in Indian culture, traces its origins to the ancient game Pachisi, which dates back to at least the 6th century (Dasgupta, 2007). Pachisi was played on large cloth boards with cowrie shells serving as dice, and it became a favorite pastime of Indian royalty. The game evolved into Ludo, which was simplified for the masses during British colonial rule in India and later spread globally (Bisht, 2018). The historical significance of Ludo is evident in its sustained popularity, which has persisted through centuries and across different cultures.

In recent years, Ludo has undergone a significant transformation from a physical board game to a digital mobile application. This digital adaptation has been driven by the broader trend of converting traditional games into mobile formats, making them more accessible and convenient for modern players (Nagata, 2021). The surge in Ludo's popularity during the COVID-19 pandemic is particularly noteworthy, as it provided a means for people to maintain social connections during lockdowns when physical interactions were severely limited (Gupta et al., 2021). Digital Ludo games, such as Ludo King and Ludo Club, became household names, offering both nostalgia and a sense of community in challenging times (Sharma and Singh, 2021).

Research on digital Ludo games has focused on various aspects, including user interface design, gameplay mechanics, and user experience. A key area of study has been the functionality of different Ludo apps, with researchers examining features such as auto-roll, token skins, and online multiplayer modes (Kumar and Banerjee, 2022). These studies have highlighted the consistency in core gameplay mechanics across different apps while also

noting the variations in additional features aimed at enhancing user engagement. Moreover, thematic analyses of user reviews have provided valuable insights into player preferences and criticisms, including issues related to game fairness, in-app advertisements, and connectivity problems (Gupta et al., 2021).

2.2. Variants of Ludo Games

Ludo has several variants, each with unique rules and gameplay mechanics that cater to different cultural and regional preferences. One of the most popular variants is *Parcheesi*, the American adaptation of Pachisi, which has become a staple in Western board gaming culture (Smith, 2015). Parcheesi introduces minor rule changes, such as the number of pieces per player and the layout of the board, making it distinct from Ludo while retaining the core mechanics of racing pieces to the finish line.

Another variant is *Uckers*, a game traditionally played in the Royal Navy, which adds elements of strategy and complexity by introducing rules like "blow-back" and "split-six" (Johnson, 2019). These additional rules make Uckers more challenging than traditional Ludo, emphasizing strategic thinking over luck. Similarly, *Fia*, a Swedish variant, incorporates additional layers of strategy by allowing players to capture each other's pieces and send them back to the start, making it a more competitive version of the game (Eriksson, 2018).

Digital variants of Ludo have also emerged, each offering unique features and gameplay experiences. For instance, *Ludo King* has introduced game modes such as Quick Ludo and Master Mode, which cater to players looking for shorter or more challenging games, respectively (Kumar and Banerjee, 2022). Other digital variants, such as *Ludo Club* and *Ludo STAR*, have incorporated social features like chat functions and team-based gameplay, enhancing the social interaction aspect of the game (Gupta et al., 2021).

2.3. Different Types of Dice

Dice have been a fundamental component of games for millennia, serving as tools for generating random outcomes in various contexts. Traditional six-sided dice, known as *cubic dice*, are the most common and are used in a wide range of games, including Ludo, Monopoly, and many others (Bell, 2004). These dice are typically made from plastic, wood, or metal, and feature pips (small dots) on each face to represent numbers from one to six.

However, not all dice are cubic. *Polyhedral dice*, which include four-sided, eight-sided, twelve-sided, and twenty-sided dice, are commonly used in tabletop role-playing games like Dungeons & Dragons (Peterson, 2012). These dice offer a broader range of outcomes and are integral to the mechanics of games that require complex probability calculations.

In recent years, *digital dice* have gained popularity, particularly in the context of mobile gaming and digital board games. These virtual dice are often incorporated into apps and games, where random number generation algorithms simulate the rolling of physical dice (Joy et al., 2021). Digital dice offer several advantages over their physical counterparts, including the ability to generate any range of numbers and integrate seamlessly into digital interfaces.

Research has also explored the fairness and randomness of dice rolls, particularly in digital environments. For example, Vassilyev et al. (2020) introduced a dice unloading algorithm that ensures fairness by counteracting inherent biases in physical dice rolls. This algorithm uses an inertial measurement unit (IMU) embedded in a die to capture real-time data, which is then processed to determine unbiased results, thus enhancing the fairness of gameplay.

2.4. Games That Include Dice

Dice-based games span a wide range of genres and have been a staple in gaming for centuries. Beyond Ludo, games like *Monopoly*, *Yahtzee*, and *Backgammon* heavily rely on dice to drive gameplay (Bell, 2004). In Monopoly, dice rolls determine the movement of players around the board, influencing their interactions with the game's economic elements. Yahtzee, a game of chance and strategy, uses five dice to create specific number combinations, rewarding players based on the outcomes of their rolls.

Backgammon, one of the oldest known board games, combines dice rolling with strategic movement of pieces on a board, where players aim to move all their pieces into their home board and then bear them off before their opponent (Murray, 1951). The randomness introduced by dice rolls in Backgammon adds an element of unpredictability, making the game a complex blend of strategy and luck.

Additionally, modern tabletop games like *Settlers of Catan* and *Risk* incorporate dice to resolve conflicts and determine resource generation, respectively (Peterson, 2012). In Settlers of Catan, dice rolls determine the resources players receive each turn, which they then use to build roads, settlements, and cities. Risk, a game of global domination, uses

dice to simulate battles between armies, with the outcome of each battle determined by the results of dice rolls.

Digital implementations of these games have also become popular, with apps and video game versions incorporating digital dice to replicate the randomness of physical dice rolls (Joy et al., 2021). These digital versions offer the convenience of automated dice rolling and calculation, streamlining gameplay and reducing the potential for human error.

2.5. Current Generation Controllers Incorporating Sensors and Microcontrollers

The current generation of gaming controllers has seen significant advancements, incorporating various sensors and microcontrollers to enhance interactivity and immersion. Modern controllers, such as those used by PlayStation, Xbox, and Nintendo consoles, are equipped with gyroscopes, accelerometers, and other sensors that allow for motion-based input (Sony Interactive Entertainment, 2023). These sensors enable features like tilt control, where players can move or tilt the controller to influence in-game actions, creating a more immersive gaming experience.

For instance, the PlayStation 5 DualSense controller includes haptic feedback and adaptive triggers, which use microcontrollers to provide dynamic resistance based on in-game events, such as the tension of pulling a bowstring or the feel of driving on different surfaces (Sony Interactive Entertainment, 2023). Similarly, the Nintendo Switch's Joy-Con controllers feature advanced motion-sensing capabilities, including an IR motion camera and HD Rumble, which provide precise feedback and allow for innovative gameplay mechanics (Nintendo, 2023).

In addition to traditional game controllers, specialized devices like the *Tobii eye tracker* and *Leap Motion controller* represent significant advancements in gaming hardware. The Tobii eye tracker uses eye-tracking technology to allow players to control games through eye movements, offering a hands-free gaming experience (Tobii Technology AB, 2023). The Leap Motion controller captures hand and finger movements, enabling players to interact with games through natural gestures without the need for a physical controller (Leap Motion, 2023).

Virtual Reality (VR) and Augmented Reality (AR) headsets, such as the Oculus Rift, represent another leap forward in gaming technology. These devices provide fully immersive experiences, where players can interact with virtual environments using motion

controllers equipped with sensors that track hand movements and orientation in 3D space (Oculus, 2023). The integration of sensors and microcontrollers in these devices allows for real-time interaction with the game world, enhancing the sense of presence and immersion in VR and AR experiences.

2.6. Innovations in Digital Dice and Puzzle Gaming

In recent years, advancements in digital gaming have led to the development of innovative devices like GoDice and GoCube by Particula, which have redefined the way traditional games are played and learned. GoDice is a set of smart, connected dice that integrate with mobile apps to offer a wide range of interactive games and educational experiences. These dice are equipped with Bluetooth connectivity, allowing them to communicate with a smartphone or tablet to provide real-time feedback and interactive gameplay (Particula, 2023).

GoDice transforms the experience of playing traditional dice games by adding a digital layer to the gameplay. The dice are embedded with sensors that detect the roll outcome, which is instantly transmitted to the connected app. This eliminates the need for manual counting or scorekeeping, making the games faster and more engaging. GoDice supports various traditional dice games like Ludo, Yahtzee, and Backgammon, as well as educational games that teach mathematics and probability in a fun and interactive way. The seamless integration with mobile apps allows for a range of customization options, including the ability to modify rules, track statistics, and even play with friends online, thus expanding the social aspect of these games (Particula, 2023).

In addition to GoDice, Particula has also developed GoCube, a smart Rubik’s Cube designed to teach and enhance the puzzle-solving experience. GoCube is equipped with a gyroscope, accelerometer, and Bluetooth chip, allowing it to track the cube's movements in real time and connect to a mobile app. This integration enables users to receive step-by-step guidance on solving the Rubik’s Cube, making it an ideal tool for beginners and enthusiasts alike (Particula, 2023).

GoCube's app provides a range of features designed to enhance learning and play. Users can follow interactive tutorials that break down the solution process into manageable steps, making it easier to learn the algorithms required to solve the cube. The app also tracks the user’s progress, provides real-time feedback on their moves, and offers competitive features such as timed challenges and leaderboards, fostering a community of puzzle

solvers (Particula, 2023). The combination of physical manipulation with digital assistance makes GoCube a unique educational tool, bridging the gap between traditional puzzles and modern technology.

The success of GoDice and GoCube illustrates the growing trend of incorporating digital enhancements into traditional games and puzzles. These products demonstrate how sensors, microcontrollers, and connectivity can transform the gaming experience, offering new ways to learn, play, and engage with classic games. As digital and physical gaming continue to converge, innovations like GoDice and GoCube are likely to inspire further developments in interactive and educational gaming.

3. Methodology

This project used an ESP-WROOM-32 (a wi-fi enabled microcontroller) along with an MPU6050 (a motion senor) to create a digital dice that can detect shaking and send the result to Unity. The communication between the ESP-WROOM-32 and computer is handled through UDP (User Datagram Protocol) over WiFi.

3.1. Creating Dice and Arduino IDE

3.1.1 Key Components

- ESP-WROOM-32: This is a powerful microcontroller that supports WiFi connectivity.
- MPU6050: A sensor that detects motion and acceleration.
- UDP (User Datagram Protocol): A communication protocol used to send data over WiFi from the dice to the computer.
- WiFi Network: The ESP-WROOM-32 connects to your home WiFi network, allowing it to communicate with other devices on the same network.

First, creating a dice functionality using ESP-WROOM-32 and MPU6050.

3.1.2 Libraries

```
#include <WiFi.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <Wire.h>
#include <WiFiUdp.h>  // Include the WiFiUdp library for UDP communication
```

- **WiFi.h:** Used to connect the ESP-WROOM-32 to your WiFi network.
- **Adafruit_MPU6050.h & Adafruit_Sensor.h:** These libraries are used to interface with the MPU6050 sensor to detect motion.
- **Wire.h:** This library handles the I2C communication protocol used to communicate with the sensor.

- **WiFiUdp.h**: This library allows communication between the ESP and the computer using the UDP protocol.

3.1.3 Wifi Setup

```
// WiFi
const char *ssid = "VM2617916";    // Enter your WiFi name
const char *password = "k2hkWtptjy7g";  // Enter WiFi password
```

- **WiFi Setup:** These lines store your WiFi network's name and password so the ESP-WROOM-32 can connect to it.

```
// UDP setup
WiFiUDP udp;
const char *udpAddress = "192.168.0.61";  // The IP address of the computer running Unity
const int udpPort = 4210;
```

- **UDP Configuration:** These variables store the IP address of the computer that will receive the dice roll results via UDP and the port number used for communication.

3.1.4 MPU Setup

```
Adafruit_MPU6050 mpu;
```

- **MPU6050 Sensor Setup:** This creates an object for interacting with the MPU6050 sensor, which will detect movement to simulate the dice roll when shaken.

```
unsigned long lastSendTime = 0;
unsigned long udpSendInterval = 500;  // Time interval to check for dice rolls (500 ms)
// Threshold to detect shaking
const float shakeThreshold = 12.0;  // Adjust this to calibrate shake sensitivity
bool diceRolled = false;            // Flag to prevent multiple rolls on one shake
```

Variables for Dice Roll Detection:

- shakeThreshold: If the sensor detects acceleration above this threshold, it interprets that as a shake, which triggers the dice roll.
- diceRolled: A flag to prevent multiple dice rolls from a single shake. This flag is reset after a short delay.

```
void setup() {
    Serial.begin(115200);

    // Initialize MPU6050
    if (!mpu.begin()) {
        Serial.println("Failed to find MPU6050 chip!");
        while (1) {
            delay(10);
        }
    }

    mpu.setAccelerometerRange(MPU6050_RANGE_8_G);
    Serial.println("MPU6050 connected!");

    // Connect to WiFi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(800);
        Serial.println("Connecting to WiFi...");
    }
    Serial.println("Connected to WiFi!");

    // Start UDP communication
    udp.begin(udpPort);
}
```

setup() Function:

1. Initialisation of the sensor: The code checks if the MPU6050 is connected and sets its sensitivity to detect shakes.
2. WiFi Connection: The ESP connects to your WiFi using the details provided earlier. It keeps trying until a successful connection is made.
3. UDP Communication: The UDP protocol is started to prepare for sending the dice roll data to the specified computer.

```
void loop() {
    sensors_event_t a, g, temp;
    mpu.getEvent(&a, &g, &temp);

    float totalAcceleration = sqrt(sq(a.acceleration.x) + sq(a.acceleration.y) + sq(a.acceleration.z));

    // Detect shaking (if the total acceleration exceeds the threshold)
    if (totalAcceleration > shakeThreshold && !diceRolled) {
        Serial.println("Shake detected!");
        int diceRoll = rollDice();  // Generate a random number (1-6)

        sendDiceRoll(diceRoll);  // Send the dice roll to Unity
        diceRolled = true;  // Set flag to prevent multiple rolls from a single shake

        // Reset the flag after a short delay to allow a new roll
        delay(2000);
        diceRolled = false;
    }

    // Send data periodically for debugging
    delay(100);
}
```

loop() Function:

- Shaking Detection: The code reads the sensor's acceleration data. If the total acceleration exceeds the shakeThreshold, it considers it a dice roll.
- Dice Roll: A random number between 1 and 6 is generated to simulate a dice roll.
- UDP Transmission: The dice roll result is sent to the specified computer using UDP.

Dice Roll and UDP Transmission Functions:

1. **rollDice():** This function generates a random number between 1 and 6 (the possible outcomes of a dice roll) and prints it to the serial monitor for debugging.

```
// Function to roll the dice (returns a random number between 1 and 6)
int rollDice() {
    int roll = random(1, 7);  // Generate random number between 1 and 6
    Serial.print("Dice rolled: ");
    Serial.println(roll);
    return roll;
}
```

2. **sendDiceRoll()**: This function sends the result of the dice roll over WiFi to the computer running the Unity application. It opens a UDP packet, sends the roll as string, and then closes the packet.

3.
```
// Function to send the dice roll via UDP
void sendDiceRoll(int roll) {
    udp.beginPacket(udpAddress, udpPort);
    udp.print(roll);  // Send the roll as a string
    udp.endPacket();
    Serial.println("Dice roll sent to Unity.");
}
```

After completed the code in Arduino IDE, I got the results in serial monitor and I have attached the image of it below:

```
Dice rolled: 3
Dice roll sent to Unity.
Shake detected!
Dice rolled: 4
Dice roll sent to Unity.
Shake detected!
Dice rolled: 1
Dice roll sent to Unity.
Shake detected!
Dice rolled: 5
Dice roll sent to Unity.
Shake detected!
Dice rolled: 2
Dice roll sent to Unity.
Shake detected!
Dice rolled: 5
Dice roll sent to Unity.
```

3.2. Creating Unity Ludo Game

3.2.1 Receiving Data From ESP-WROOM-32 and MPU6050

In the next step, I started working in unity to receive the data which was being sent from ESP-WROOM-32 about MPU6050.

1. Imports and Class Declaration

```
using System;
using System.Text;
using UnityEngine;
using UnityEngine.UI;
using System.Net;
using System.Net.Sockets;
using System.Threading;

public class LudoDice : MonoBehaviour
```

- **Imports:** These are libraries that provide functionalities like networking (System.Net, System.Net.Sockets), threading (System.Threading), and Unity-specific functions (UnityEngine).
- **Class Declaration:** The class LudoDice inherits from Unity's MonoBehaviour, meaning it can be attached to objects in Unity and interact with the game engine.

2. Class Variables

```
private Thread receiveThread;
private UdpClient client;
public int port = 4210;  // UDP Port to listen on (same as ESP32)
public string receivedData;

private volatile bool isReceiving = false;  // Use 'volatile' for thread-safe control
```

- **receiveThread:** This is the thread that will handle receiving data from the ESP32.
- **UdpClient:** This is used to manage the UDP connection.
- **port:** The port the UDP client will listen on (same as the one the ESP32 sends data to).
- **receivedData:** This stores the dice roll data received from the ESP32.
- **isReceiving:** A boolean flag to control whether the script is currently receiving data. The keyword volatile ensures that changes to this variable are visible to all threads.

3. Start() Method

```
void Start()
{
    StartUdpListener();  // Start listening for UDP messages
}
```

- **Purpose:** This method is automatically called when the Unity game object with this script starts. It calls StartUdpListener() to begin receiving UDP data.

4. StartUdpListener() Method

```
void StartUdpListener()
{
    isReceiving = true;  // Set the flag to indicate that data is being received
    receiveThread = new Thread(new ThreadStart(ReceiveData));
    receiveThread.IsBackground = true;  // Set the thread to run in the background
    receiveThread.Start();
    Debug.Log($"Listening for UDP packets on port {port}.");
}
```

- **Start UDP Listening:** This method sets up a new background thread to handle receiving UDP data without interrupting the main game thread. It logs a message to the Unity console to indicate that it's listening for UDP messages.

5. ReceiveData() Method

```
void ReceiveData()
{
    try
    {
        client = new UdpClient(port);
        IPEndPoint anyIP = new IPEndPoint(IPAddress.Any, port);

        while (isReceiving)  // Keep receiving data while the flag is true
        {
            try
            {
                byte[] data = client.Receive(ref anyIP);  // Receive UDP data
                string text = Encoding.UTF8.GetString(data);
                receivedData = text.Trim();  // Store received dice roll

                Debug.Log($"Received dice roll: {receivedData}");
            }
            catch (SocketException ex)
            {
                if (ex.SocketErrorCode != SocketError.Interrupted)
                {
                    Debug.LogError($"UDP Receive error: {ex.Message}");
                }
            }
        }
    }
    catch (Exception err)
    {
        Debug.LogError($"Error setting up UDP client: {err.Message}");
    }
}
```

- **Set Up UDP Client:** This method creates the UDP client and listens on the specified port.
- **Receiving Data:** The method stays in a loop, constantly receiving data as long as isReceiving is true. It converts the received byte array into a string (which is the dice roll value) and stores it in receivedData.

- **Error Handling:** If there is an issue (e.g., network failure), it logs the error.

6. Update() Method

```
void Update()
{
    if (!string.IsNullOrEmpty(receivedData))  // If new data is received
    {
        int diceRoll = int.Parse(receivedData);  // Parse the dice roll result from received data

        // Call the GameScript's DiceRoll method with the dice roll result
        FindObjectOfType<GameScript>().DiceRoll(diceRoll);

        receivedData = "";  // Reset the received data after handling it
    }
}
```

- **Process the Received Data:** This method runs once per frame. It checks if new data has been received (receivedData). If so, it converts the string data into an integer (representing the dice roll) and calls a method from another script (GameScript) to handle the dice roll in the game.
- **Clear Data:** After processing, it clears receivedData to be ready for the next incoming message.

7. OnApplicationQuit() Method

```
void OnApplicationQuit()
{
    StopUdpListener();  // Stop the UDP listener when the application quits
}
```

- **Clean Up**: This method is called when the application is about to quit. It ensures the UDP listener is properly stopped and the resources are released.

8. StopUdpListener() Method

```
void StopUdpListener()
{
    isReceiving = false;  // Set the flag to stop receiving data

    // Wait for the thread to terminate gracefully
    if (receiveThread != null && receiveThread.IsAlive)
    {
        receiveThread.Join();  // Block until the thread finishes execution
    }

    // Close the UDP client to release the port
    if (client != null)
    {
        client.Close();
    }

    Debug.Log("UDP listener stopped.");
}
```

- **Stop UDP Listening**: This method stops the UDP listener by setting isReceiving to false, which exits the receiving loop. It waits for the background thread to finish execution, closes the UDP client, and logs that the listener has stopped.

3.2.2 Unity Main Thread Dispatcher

Also, I had to add Unity Main Thread Dispatcher as while connecting the ESP-WROOM-32 with Unity. By adding Unity Main Thread Dispatcher, I was able to avoid the connection loss like if I stop the game in Unity and play again then I had to reset the dice but after adding Unity Main Thread Dispatcher it was fixed.

1. Execution Queue and Singleton Instance

```
private static readonly Queue<Action> _executionQueue = new Queue<Action>();

private static UnityMainThreadDispatcher _instance;
```

- **_executionQueue:** This is a queue that stores tasks (Actions) to be executed on the main thread. A queue works like a line: actions get added to the back and processed from the front, one at a time.
- **Singleton Pattern (_instance):** This variable holds the single instance of the UnityMainThreadDispatcher class. The singleton pattern ensures that there's only one instance of this dispatcher running in the game, preventing conflicts or duplicate operations.

2. Instance() Method

```
public static UnityMainThreadDispatcher Instance()
{
    if (!_instance)
    {
        // Find an existing dispatcher or create a new one
        _instance = FindObjectOfType<UnityMainThreadDispatcher>();
        if (!_instance)
        {
            GameObject obj = new GameObject("UnityMainThreadDispatcher");
            _instance = obj.AddComponent<UnityMainThreadDispatcher>();
            DontDestroyOnLoad(obj);  // Persist across scene loads
        }
    }
    return _instance;
}
```

Purpose: This method ensures there's always an instance of UnityMainThreadDispatcher running in your game.

- It first checks if _instance already exists.
- If it doesn't exist, it tries to find an existing dispatcher in the scene using FindObjectOfType.
- If it still doesn't find one, it creates a new GameObject called "UnityMainThreadDispatcher" and attaches this script to it as a component.
- The line DontDestroyOnLoad(obj) ensures the dispatcher persists across different scenes.

3. Enqueue() Method

```
// Enqueue an action to be executed on the main Unity thread
public void Enqueue(Action action)
{
    lock (_executionQueue)
    {
        _executionQueue.Enqueue(action);
    }
}
```

Purpose: This method allows you to add (or “enqueue”) an action to the queue, ensuring it will be executed on the main thread.

- The lock keyword ensures that only one thread can add actions to the queue at a time, avoiding issues where multiple threads might try to modify the queue simultaneously.

4. Update() Method

```
void Update()
{
    lock (_executionQueue)
    {
        while (_executionQueue.Count > 0)
        {
            _executionQueue.Dequeue().Invoke();
        }
    }
}
```

Purpose: This method runs every frame in Unity (as part of the MonoBehaviour lifecycle). It processes the actions in the queue and executes them on the main thread.

- The **lock** ensures that while the main thread is processing actions, no other thread can modify the queue.
- **Dequeue()** removes an action from the front of the queue, and **Invoke()** executes that action.

With the help of both the script, I was able to sent data from ESP-WROOM-32 to Unity. Also, I had to disable Windows firewall as it was blocking the connection.

3.2.3 Creating Main Menu

1. Static Variable Declaration

```
public static int howManyPlayers;
```

Purpose: This variable is used to store how many players will be playing the game. Since it's static, it is shared across all instances of this class and persists even when switching scenes. This allows the selected number of players to be accessed in the gameplay scene.

2. two_player() Method

```
public void two_player()
{
    SoundManagerScript.buttonAudioSource.Play ();
    howManyPlayers = 2;
    SceneManager.LoadScene ("Ludo");
}
```

Purpose: This method is triggered when the player selects the option to play with two players.

- **Play Button Sound**: The SoundManagerScript.buttonAudioSource.Play() line plays a sound when the button is clicked.
- **Set Players**: howManyPlayers = 2; sets the static variable to 2, meaning the game will have two players.

- **Load Scene**: The SceneManager.LoadScene("Ludo") loads the scene named "Ludo", which is presumably where the actual gameplay happens.

3. three_player() Method

```
public void three_player()
{
    SoundManagerScript.buttonAudioSource.Play ();
    howManyPlayers = 3;
    SceneManager.LoadScene ("Ludo");
}
```

Purpose: This method is almost identical to two_player() but sets the number of players to 3 instead. When a player selects three players, it:

- Plays the button click sound.
- Sets howManyPlayers to 3.
- Loads the "Ludo" gameplay scene.

4. four_player() Method

```
public void four_player()
{
    SoundManagerScript.buttonAudioSource.Play ();
    howManyPlayers = 4;
    SceneManager.LoadScene ("Ludo");
}
```

Purpose: Similar to the previous two methods, this sets the game to have four players when the user selects that option. It:

- Plays the button click sound.
- Sets howManyPlayers to 4.
- Loads the "Ludo" gameplay scene.

5. quit() Method

```
public void quit()
{
    SoundManagerScript.buttonAudioSource.Play ();
    Application.Quit ();
}
```

Purpose: This method is used to exit the game.

- It plays the button sound before quitting the application.
- Application.Quit() closes the game when the player selects the quit option (Note: this will only work when the game is running as a standalone build, not in the Unity editor).

3.2.4 Adding Sounds

1. **Public Variables for Audio Clips**

```
public AudioClip buttonAudioClip;
public AudioClip dismissalAudioClip;
public AudioClip diceAudioClip;
public AudioClip winAudioClip;
public AudioClip safeHouseAudioClip;
public AudioClip playerAudioClip;
```

Purpose: These are placeholders for different sound effects that the game will use. They are declared as public so that audio clips can be assigned to them in Unity's Inspector window.

- **buttonAudioClip:** Sound played when a button is clicked.
- **dismissalAudioClip:** Sound played when a player is dismissed in the game.
- **diceAudioClip:** Sound played when rolling the dice.
- **winAudioClip:** Sound played when a player wins.
- **safeHouseAudioClip:** Sound played when a player reaches a safe house.
- **playerAudioClip:** General sound associated with the player.

2. **Static AudioSource Variables**

```
public static AudioSource buttonAudioSource;
public static AudioSource dismissalAudioSource;
public static AudioSource diceAudioSource;
public static AudioSource winAudioSource;
public static AudioSource safeHouseAudioSource;
public static AudioSource playerAudioSource;
```

Purpose: These variables are used to control the audio playback for each sound effect. Since they are declared as static, they can be accessed by other scripts in the game. Each AudioSource corresponds to the audio clips defined earlier.

- **buttonAudioSource:** Controls the playback of the button sound.
- **dismissalAudioSource:** Controls the dismissal sound.
- **diceAudioSource:** Controls the dice sound.
- **winAudioSource:** Controls the winning sound.
- **safeHouseAudioSource:** Controls the safe house sound.
- **playerAudioSource:** Controls player-related sounds.

4. AddAudio() Method

```
AudioSource AddAudio(AudioClip clip, bool playOnAwake, bool loop, float  volume)
{
    AudioSource audioSource = gameObject.AddComponent<AudioSource> ();
    audioSource.clip = clip;
    audioSource.playOnAwake = playOnAwake;
    audioSource.loop = loop;
    audioSource.volume = volume;
    return audioSource;
}
```

- **Purpose:** This method is used to create and configure an AudioSource component for each sound effect. It returns the AudioSource that is attached to the game object.
    - **clip:** The sound file to be played.
    - **playOnAwake:** If true, the sound will play as soon as the scene starts. Here, it's false because the sounds only play on specific actions (like button clicks or dice rolls).
    - **loop:** If true, the sound will repeat continuously. In this game, sounds like button clicks or dice rolls likely won't loop, so this is set to false.
    - **volume:** The volume of the sound. This is set to 1.0f, which is full volume.

The method adds an AudioSource component to the game object this script is attached to, then sets up its properties based on the parameters provided.

5. Start() Method

```
void Start ()
{
    buttonAudioSource = AddAudio (buttonAudioClip,false, false, 1.0f);
    dismissalAudioSource = AddAudio (dismissalAudioClip,false, false, 1.0f);
    diceAudioSource = AddAudio (diceAudioClip,false, false, 1.0f);
    winAudioSource = AddAudio (winAudioClip,false, false, 1.0f);
    safeHouseAudioSource = AddAudio (safeHouseAudioClip,false, false, 1.0f);
    playerAudioSource = AddAudio (playerAudioClip,false, false, 1.0f);
}
```

Purpose: The Start() method is called automatically when the game begins. It initialises all the AudioSource components by calling the AddAudio() method for each sound effect.

3.2.5 Creating players for game

As there are 4 players in the game and each player has 4 tokens which needs to be taken to house safely to win the game. Also, all the players had the same script as it was collider detecting if the player has reached home or not.

1. **Static Variable Declaration**

```
public static string bluePlayerI_ColName;
```

Purpose: This static variable holds the name of the object that the blue player piece collides with. Since it's static, it means the variable is shared among all instances of this class and can be accessed from other scripts if needed.

- For example, it might store the name of a block or safe house the blue player has landed on.

2. **OnTriggerEnter2D() Method**

```
void OnTriggerEnter2D(Collider2D col)
{
    if (col.gameObject.tag == "blocks")
    {
        bluePlayerI_ColName = col.gameObject.name;
        if (col.gameObject.name.Contains("Safe House")) {
            SoundManagerScript.safeHouseAudioSource.Play ();
        }
    }
}
```

Purpose: This method is triggered automatically when the blue player piece enters a collision with another object (that has a 2D Collider).

- **Collision with Blocks:** The first check is to see if the object the player collides with has the tag "blocks". This ensures that only collisions with certain game objects (like board blocks) are considered.
- **Storing the Block Name:** When the blue player collides with a block, the block's name is stored in the bluePlayerI_ColName variable. This could be used later to check where the player is on the game board.
- **Safe House Check:** The code also checks if the block's name contains the phrase "Safe House". If it does, this means the player has landed on a safe house, and the script plays a safe house sound using SoundManagerScript.safeHouseAudioSource.Play();.

3. **Start() Method**

```
void Start ()
{
    bluePlayerI_ColName = "none";
}
```

Purpose: This method is called automatically when the game object is first created or the scene starts.

- It sets the bluePlayerI_ColName to "none" at the start of the game. This acts as a default value, indicating that the player hasn't collided with anything yet.

3.2.6 Creating Game Script

This script would handle most of the core logic for Ludo Game, including players turns, dice rolling, movement, and game completion.

1. **Imports and Class Declaration**

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

Imports: These are libraries needed for different functionalities:

- **UnityEngine:** The core Unity library used to access game objects, components, and the game engine.
- **System.Collections and System.Collections.Generic:** These are used for handling lists and collections.
- **UnityEngine.UI:** Provides access to UI elements like buttons and text.
- **SceneManagement:** Allows switching between different game scenes.
- **Class Declaration:** The GameScript class inherits from Unity's MonoBehaviour, meaning it can be attached to game objects and interact with the game engine.

2. Variable Declarations

```
private int totalBlueInHouse, totalRedInHouse, totalGreenInHouse, totalYellowInHouse;

public GameObject frameRed, frameGreen, frameBlue, frameYellow;

public GameObject redPlayerI_Border, redPlayerII_Border, redPlayerIII_Border, redPlayerIV_Border;
public GameObject greenPlayerI_Border, greenPlayerII_Border, greenPlayerIII_Border, greenPlayerIV_Border;
public GameObject bluePlayerI_Border, bluePlayerII_Border, bluePlayerIII_Border, bluePlayerIV_Border;
public GameObject yellowPlayerI_Border, yellowPlayerII_Border, yellowPlayerIII_Border, yellowPlayerIV_Border;
```

Purpose: The script defines variables that track important game elements:

- **Player Houses**: totalBlueInHouse, totalRedInHouse, etc., keep track of how many players have reached their respective "houses" or safe areas.
- **Frames**: frameRed, frameGreen, etc., likely highlight the current player's turn.
- **Borders**: Each player piece (Red, Green, Blue, and Yellow) has four borders, likely used to show which piece is selectable for movement.

3. Player Position and Button Handling

```
public Vector3 redPlayerI_Pos, redPlayerII_Pos, redPlayerIII_Pos, redPlayerIV_Pos;
public Vector3 greenPlayerI_Pos, greenPlayerII_Pos, greenPlayerIII_Pos, greenPlayerIV_Pos;
public Vector3 bluePlayerI_Pos, bluePlayerII_Pos, bluePlayerIII_Pos, bluePlayerIV_Pos;
public Vector3 yellowPlayerI_Pos, yellowPlayerII_Pos, yellowPlayerIII_Pos, yellowPlayerIV_Pos;

public Button RedPlayerI_Button, RedPlayerII_Button, RedPlayerIII_Button, RedPlayerIV_Button;
public Button GreenPlayerI_Button, GreenPlayerII_Button, GreenPlayerIII_Button, GreenPlayerIV_Button;
public Button BluePlayerI_Button, BluePlayerII_Button, BluePlayerIII_Button, BluePlayerIV_Button;
public Button YellowPlayerI_Button, YellowPlayerII_Button, YellowPlayerIII_Button, YellowPlayerIV_Button;
```

- **Player Positions**: Each player piece (Red, Green, Blue, and Yellow) has its starting position stored as a Vector3 (3D coordinates in the game world).
- **UI Buttons**: Each player piece has an associated button in the UI, used for selecting the piece during the game.

4. Player Turn Management

```
private string playerTurn = "RED";
public Transform diceRoll;
public Button DiceRollButton;
public Transform redDiceRollPos, greenDiceRollPos, blueDiceRollPos, yellowDiceRollPos;

private string currentPlayer = "none";
private string currentPlayerName = "none";
```

- **playerTurn**: Keeps track of whose turn it is. It starts with "RED".
- **diceRoll and DiceRollButton**: The dice roll UI element and the button used to roll the dice.
- **currentPlayer and currentPlayerName**: Track which player piece is currently being moved.

5. Dice Animation Setup

```
//--------------- Dice Animations------
public GameObject dice1_Roll_Animation;
public GameObject dice2_Roll_Animation;
public GameObject dice3_Roll_Animation;
public GameObject dice4_Roll_Animation;
public GameObject dice5_Roll_Animation;
public GameObject dice6_Roll_Animation;
```

- **Dice Animations**: Each possible dice result (1 through 6) has an associated animation that is displayed when the dice is rolled**.**

6. Player Movement

```
// Players movement corenspoding to blocks...
public List<GameObject> redMovementBlocks = new List<GameObject>();
public List<GameObject> greenMovementBlocks = new List<GameObject>();
public List<GameObject> yellowMovementBlocks = new List<GameObject>();
public List<GameObject> blueMovementBlocks = new List<GameObject>();
```

- **Movement Blocks**: Each player has a list of blocks they must move through on the board. This is tracked using lists for Red, Green, Yellow, and Blue.

7. Dice Roll Handling

```
public void DiceRoll(int diceRollResult)
{
    DiceRollButton.interactable = false;
    selectDiceNumAnimation = diceRollResult;  // Use the provided dice roll result
    PlayDiceAnimation(selectDiceNumAnimation);
    StartCoroutine(PlayersNotInitialized());
}
```

Dice Roll: This method handles what happens when the dice is rolled. The result of the roll (diceRollResult) is passed in, and the corresponding animation is played. Then, it checks if the players are ready for movement.

8. Game Completion Logic

```
// -============== GAME COMPLETED ROUTINE =====
IEnumerator GameCompletedRoutine()
{
    yield return new WaitForSeconds(1.5f);
    gameCompletedScreen.SetActive(true);
}
```

Game Completion: When a player wins, the game waits for 1.5 seconds and then activates the game completed screen.

9. Resetting the Dice Roll State

```
// DICE Initialization after players have finished their turn---------------
void InitializeDice()
{
    DiceRollButton.interactable = true;

    dice1_Roll_Animation.SetActive(false);
    dice2_Roll_Animation.SetActive(false);
    dice3_Roll_Animation.SetActive(false);
    dice4_Roll_Animation.SetActive(false);
    dice5_Roll_Animation.SetActive(false);
    dice6_Roll_Animation.SetActive(false);
```

Purpose: This block of code resets the dice roll interface.

- Enable Dice Roll Button: The first line makes sure that the button for rolling the dice is clickable by setting it as interactable.

- Deactivate All Dice Roll Animations: All six possible dice roll animations are turned off to prepare for the next roll. This ensures that no previous animations are displayed when the player rolls the dice again.

10. Checking for Game Completion (Two Players)

```
switch (MainMenuScript.howManyPlayers)
{
    case 2:
        if (totalRedInHouse > 3)
        {
            SoundManagerScript.winAudioSource.Play();
            redScreen.SetActive(true);
            StartCoroutine("GameCompletedRoutine");
            playerTurn = "NONE";
        }

        if (totalGreenInHouse > 3)
        {
            SoundManagerScript.winAudioSource.Play();
            greenScreen.SetActive(true);
            StartCoroutine("GameCompletedRoutine");
            playerTurn = "NONE";
        }
        break;
```

Purpose: This section checks whether any player has won when there are 2 players in the game.

- Winning Condition: A player wins if all their pieces (greater than 3) have made it into the house (i.e., the safe area on the board).
- Winning Actions: When a player wins, the victory sound is played, the corresponding screen is shown (e.g., redScreen for Red Player), and the game completes using a GameCompletedRoutine. The playerTurn is set to "NONE" to prevent further actions.

11. Handling Game Completion (Three Players)

```
case 3:
    // If any 1 of 3 player wins==============================================
    if (totalRedInHouse > 3 && totalBlueInHouse < 4 && totalYellowInHouse < 4 && playerTurn == "RED")
    {
        if (!redScreen.activeInHierarchy)
        {
            SoundManagerScript.winAudioSource.Play();
        }

        redScreen.SetActive(true);
        Debug.Log("Red Player WON");
        playerTurn = "BLUE";
    }
```

Purpose: This block handles cases where 3 players are in the game.

- Sequential Winning: The script checks if each player has won based on whose turn it is and whether the other players still have active pieces.

- Next Turn: After a player wins (e.g., Red), the turn is passed to the next player (e.g., Blue).

12. Handling Game Completion (Four Players)

```
case 4:
    // If any 1 of 4 player wins==========================================================================
    if (totalRedInHouse > 3 && totalBlueInHouse < 4 && totalGreenInHouse < 4 && totalYellowInHouse < 4 && playerTurn == "RED")
    {
        if (!redScreen.activeInHierarchy)
        {
            SoundManagerScript.winAudioSource.Play();
        }

        redScreen.SetActive(true);
        Debug.Log("Red Player WON");
        playerTurn = "BLUE";
    }
```

Purpose: When there are 4 players, this block checks if any player has won.

- Victory Logic: The script checks if all pieces of a player (e.g., Red Player) have entered the house, ensuring the other players (e.g., Blue, Green, Yellow) have not finished their game.
- Passing Turns: Similar to the 3-player scenario, once a player wins, the game passes to the next player.

13. Tracking the Current Player's Position

```
//======== Getting currentPlayer VALUE=======
if (currentPlayerName.Contains("RED PLAYER"))
{
    if (currentPlayerName == "RED PLAYER I")
    {
        Debug.Log("currentPlayerName = " + currentPlayerName);
        currentPlayer = RedPlayerI_Script.redPlayerI_ColName;
    }
    if (currentPlayerName == "RED PLAYER II")
    {
        Debug.Log("currentPlayerName = " + currentPlayerName);
        currentPlayer = RedPlayerII_Script.redPlayerII_ColName;
    }
    if (currentPlayerName == "RED PLAYER III")
    {
        Debug.Log("currentPlayerName = " + currentPlayerName);
        currentPlayer = RedPlayerIII_Script.redPlayerIII_ColName;
    }
    if (currentPlayerName == "RED PLAYER IV")
    {
        Debug.Log("currentPlayerName = " + currentPlayerName);
        currentPlayer = RedPlayerIV_Script.redPlayerIV_ColName;
    }
}
```

Purpose: This block checks which player piece is currently being moved and stores its position.

- Identifying the Current Piece: It uses the name of the current player's piece (e.g., RED PLAYER I) to update its current position, using the respective script (e.g., RedPlayerI_Script).

- Dynamic Assignment: This ensures that the movement or actions apply to the correct player piece depending on who is playing.

14. Player vs Opponent Logic

```
if (currentPlayer == GreenPlayerI_Script.greenPlayerI_ColName && (currentPlayer != "Star" && GreenPlayerI_Script.greenPlayerI_ColName != "Star"))
{
    SoundManagerScript.dismissalAudioSource.Play();
    greenPlayerI.transform.position = greenPlayerI_Pos;
    GreenPlayerI_Script.greenPlayerI_ColName = "none";
    greenPlayerI_Steps = 0;
    playerTurn = "RED";
```

Purpose: This block handles when a player's piece collides with an opponent's piece.

- Opponent Knockout: If a Red player's piece lands on the same block as a Green player's piece (except for special "Star" blocks), the Green player's piece is sent back to its starting position.
- Sound and Reset: The dismissal sound plays, the defeated piece is reset to its starting position, and the number of steps the piece has taken is reset to 0. The turn is then given back to the attacking player.

15. Coroutine Declaration and Initial Wait

```
IEnumerator PlayersNotInitialized()
{
    yield return new WaitForSeconds(0.8f);
```

- Coroutine: In Unity, a coroutine allows you to execute code over several frames. This is useful for timing events without freezing the game.
- WaitForSeconds(0.8f): The coroutine pauses execution for 0.8 seconds before proceeding. This delay can be used to ensure that all animations or game states are properly updated before checking for player moves.

16. Determining the Current Player's Turn

```
case "RED":

    //================== CONDITION FOR BORDER GLOW ======================
    if ((redMovementBlocks.Count - redPlayerI_Steps) >= selectDiceNumAnimation && redPlayerI_Steps > 0 && (redMovementBlocks.Count > redPlayerI_Steps))
    {
        redPlayerI_Border.SetActive(true);
        RedPlayerI_Button.interactable = true;
    }
    else
    {
        redPlayerI_Border.SetActive(false);
        RedPlayerI_Button.interactable = false;
    }
```

- playerTurn: A string variable that keeps track of whose turn it is (e.g., "RED", "BLUE", "GREEN", "YELLOW").
- Switch Statement: Depending on the value of playerTurn, the coroutine executes different blocks of code tailored to each player's turn.

17. Handling Each Player's Turn

Case "RED": Managing Red Player's Turn

```
case "RED":

    //================== CONDITION FOR BORDER GLOW ======================
    if ((redMovementBlocks.Count - redPlayerI_Steps) >= selectDiceNumAnimation && redPlayerI_Steps > 0 && (redMovementBlocks.Count > redPlayerI_Steps))
    {
        redPlayerI_Border.SetActive(true);
        RedPlayerI_Button.interactable = true;
    }
    else
    {
        redPlayerI_Border.SetActive(false);
        RedPlayerI_Button.interactable = false;
    }

    if ((redMovementBlocks.Count - redPlayerII_Steps) >= selectDiceNumAnimation && redPlayerII_Steps > 0 && (redMovementBlocks.Count > redPlayerII_Steps))
    {
        redPlayerII_Border.SetActive(true);
        RedPlayerII_Button.interactable = true;
    }
    else
    {
        redPlayerII_Border.SetActive(false);
        RedPlayerII_Button.interactable = false;
    }

    if ((redMovementBlocks.Count - redPlayerIII_Steps) >= selectDiceNumAnimation && redPlayerIII_Steps > 0 && (redMovementBlocks.Count > redPlayerIII_Steps))
    {
        redPlayerIII_Border.SetActive(true);
        RedPlayerIII_Button.interactable = true;
    }
    else
    {
        redPlayerIII_Border.SetActive(false);
        RedPlayerIII_Button.interactable = false;
    }
```

```
if (!redPlayerI_Border.activeInHierarchy && !redPlayerII_Border.activeInHierarchy &&
    !redPlayerIII_Border.activeInHierarchy && !redPlayerIV_Border.activeInHierarchy)
{
    RedPlayerI_Button.interactable = false;
    RedPlayerII_Button.interactable = false;
    RedPlayerIII_Button.interactable = false;
    RedPlayerIV_Button.interactable = false;

    switch (MainMenuScript.howManyPlayers)
    {
        case 2:
            playerTurn = "GREEN";
            InitializeDice();
            break;

        case 3:
            playerTurn = "BLUE";
            InitializeDice();
            break;

        case 4:
            playerTurn = "BLUE";
            InitializeDice();
            break;
    }
}
break;
```

Purpose: Determines whether a specific player piece (e.g., Red Player I) can make a move based on the dice roll and its current position.

Variables:

- **redMovementBlocks.Count:** Total number of movement blocks (positions) available to the Red player.
- **redPlayerI_Steps:** Number of steps the Red Player I has already taken.
- **selectDiceNumAnimation:** The result of the dice roll determining how many steps the player can move.

Logic:

- **Movement Possibility:** Checks if moving the player piece by the number rolled (selectDiceNumAnimation) keeps it within the bounds of the game board.
- **Player Already in Play:** Ensures the player piece has already entered the game (i.e., redPlayerI_Steps > 0).
- **Boundary Check:** Ensures the player piece isn't trying to move beyond the last movement block.

Actions:

If Conditions Met:

- **redPlayerI_Border.SetActive(true):** Activates a visual border around the player piece to indicate it can be selected.
- **RedPlayerI_Button.interactable = true:** Makes the UI button for this player piece clickable.

If Conditions Not Met:

- **redPlayerI_Border.SetActive(false):** Deactivates the visual border.
- **RedPlayerI_Button.interactable = false:** Disables the UI button to prevent selection.
- **Repetition:** Similar conditional blocks are present for other Red player pieces (redPlayerII, redPlayerIII, redPlayerIV), ensuring each piece is individually assessed for movement.

Special Condition When Rolling a Six

```
if (selectDiceNumAnimation == 6 && redPlayerI_Steps == 0)
{
    redPlayerI_Border.SetActive(true);
    RedPlayerI_Button.interactable = true;
}
```

Purpose: In many board games like Ludo, rolling a six grants the player the ability to bring a new piece into play. This condition checks if the player rolled a six and if a particular piece is still at the starting position (i.e., redPlayerI_Steps == 0).

Actions:

- **redPlayerI_Border.SetActive(true):** Highlights the player piece to indicate it can be moved onto the board.
- **RedPlayerI_Button.interactable = true:** Enables the button for the player to select this piece and move it into play.

17.3 Handling No Available Moves and Switching Turns

```
if (!redPlayerI_Border.activeInHierarchy && !redPlayerII_Border.activeInHierarchy &&
    !redPlayerIII_Border.activeInHierarchy && !redPlayerIV_Border.activeInHierarchy)
{
    RedPlayerI_Button.interactable = false;
    RedPlayerII_Button.interactable = false;
    RedPlayerIII_Button.interactable = false;
    RedPlayerIV_Button.interactable = false;

    switch (MainMenuScript.howManyPlayers)
    {
        case 2:
            playerTurn = "GREEN";
            InitializeDice();
            break;

        case 3:
            playerTurn = "BLUE";
            InitializeDice();
            break;

        case 4:
            playerTurn = "BLUE";
            InitializeDice();
            break;
    }
}
break;
```

Purpose: Checks if none of the Red player's pieces can make a move based on the current dice roll.

Conditions:

- !redPlayerI_Border.activeInHierarchy: Red Player I cannot move.
- Similar checks for Red Player II, III, and IV.

Actions:

1. **Disable All Red Player Buttons:** Ensures no Red player piece is selectable since none can move.
2. **Switch Player Turn:**

- **2 Players:** If there are two players, the turn switches to "GREEN".
- **3 Players:** The turn switches to "BLUE".
- **4 Players:** Also switches to "BLUE".

3. **Initialize Dice:** Calls the InitializeDice() method to reset the dice state for the next player's turn.

18. Frame Rate and V-Sync Setup

```
void Start()

{
    QualitySettings.vSyncCount = 1;
    Application.targetFrameRate = 30;
```

Purpose: These lines control the performance settings of the game.

- **V-Sync:** Setting vSyncCount to 1 ensures that the game synchronises the frame rate with the display's refresh rate, preventing screen tearing.
- **Frame Rate:** The target frame rate is set to 30 frames per second. This is likely done to ensure that the game runs smoothly and consistently, especially on devices with lower performance.

19. Random Number Generator Initialization

```
randomNo = new System.Random();
```

Purpose: A random number generator is initialized. This will likely be used later in the game to handle the rolling of dice or other random events.

3. Disabling Dice Roll Animations

```
dice1_Roll_Animation.SetActive(false);
dice2_Roll_Animation.SetActive(false);
dice3_Roll_Animation.SetActive(false);
dice4_Roll_Animation.SetActive(false);
dice5_Roll_Animation.SetActive(false);
dice6_Roll_Animation.SetActive(false);
```

Purpose: All dice roll animations (from rolling a 1 to 6) are deactivated at the start of the game to ensure that no dice animations are showing when the game begins.

4. Storing Initial Player Positions

```
// Players initial positions.....
redPlayerI_Pos = redPlayerI.transform.position;
redPlayerII_Pos = redPlayerII.transform.position;
redPlayerIII_Pos = redPlayerIII.transform.position;
redPlayerIV_Pos = redPlayerIV.transform.position;

greenPlayerI_Pos = greenPlayerI.transform.position;
greenPlayerII_Pos = greenPlayerII.transform.position;
greenPlayerIII_Pos = greenPlayerIII.transform.position;
greenPlayerIV_Pos = greenPlayerIV.transform.position;
```

Purpose: The initial positions of all player pieces (Red, Green, Blue, Yellow) are stored. These positions will serve as reference points in the game, possibly used to reset players to their starting spots if necessary.

Player Positions: Each player has four pieces, and for each piece (e.g., redPlayerI, greenPlayerII), their starting position is recorded using transform.position.

5. Deactivating Player Borders

```
redPlayerI_Border.SetActive(false);
redPlayerII_Border.SetActive(false);
redPlayerIII_Border.SetActive(false);
redPlayerIV_Border.SetActive(false);
```

Purpose: All borders (visual indicators around the player pieces) are deactivated at the start of the game. These borders are likely used later during gameplay to highlight which pieces can be moved or selected.

6. Hiding the Winning Screens

```
redScreen.SetActive(false);
greenScreen.SetActive(false);
yellowScreen.SetActive(false);
blueScreen.SetActive(false);
```

Purpose: The winning screens for each player are hidden at the start of the game. These screens likely appear when a player wins the game, so they are turned off until a player wins.

7. Setting Up the Game Based on the Number of Players

Case 1: Two Players

```
switch (MainMenuScript.howManyPlayers)
{
    case 2:
        playerTurn = "RED";

        frameRed.SetActive(true);
        frameGreen.SetActive(false);
        frameBlue.SetActive(false);
        frameYellow.SetActive(false);
        //diceRoll.position = redDiceRollPos.position;
        bluePlayerI.SetActive(false);
        bluePlayerII.SetActive(false);
        bluePlayerIII.SetActive(false);
        bluePlayerIV.SetActive(false);

        yellowPlayerI.SetActive(false);
        yellowPlayerII.SetActive(false);
        yellowPlayerIII.SetActive(false);
        yellowPlayerIV.SetActive(false);
        break;
```

- **Player Turn:** The game is set to start with the Red player's turn (playerTurn = "RED").
- **Visual Frames:** The Red player's frame (a visual cue to show it's their turn) is activated (frameRed.SetActive(true)), while the frames for the other players are deactivated.
- **Deactivating Unused Players:** Since there are only two players (Red and Green), the Blue and Yellow player pieces are deactivated (SetActive(false)), meaning they won't be part of the game.

Case 2: Three Players

```
case 3:
    playerTurn = "YELLOW";

    frameRed.SetActive(false);
    frameGreen.SetActive(false);
    frameBlue.SetActive(false);
    frameYellow.SetActive(true);

    diceRoll.position = yellowDiceRollPos.position;
    greenPlayerI.SetActive(false);
    greenPlayerII.SetActive(false);
    greenPlayerIII.SetActive(false);
    greenPlayerIV.SetActive(false);

    break;
```

- **Player Turn:** For three players, the game starts with Yellow's turn (playerTurn = "YELLOW").

- **Visual Frames:** The Yellow player's frame is shown, indicating it's their turn, while the other frames are deactivated.
- **Dice Position:** The position of the dice roll UI is moved to match the Yellow player (diceRoll.position = yellowDiceRollPos.position), likely providing a visual cue near the Yellow player pieces.
- **Deactivating Unused Players:** Since only Red, Blue, and Yellow are playing, all Green player pieces are deactivated.

Case 3: Four Players

```
case 4:
    playerTurn = "RED";

    frameRed.SetActive(true);
    frameGreen.SetActive(false);
    frameBlue.SetActive(false);
    frameYellow.SetActive(false);

    diceRoll.position = redDiceRollPos.position;
    // keep all players active
    break;
```

- **Player Turn:** In a four-player game, Red starts first (playerTurn = "RED").
- **Visual Frames:** Only the Red player's frame is shown at the start.
- **Dice Position:** The dice roll UI is positioned near the Red player (diceRoll.position = redDiceRollPos.position).
- **Keeping All Players Active:** Since it's a four-player game, all player pieces (Red, Green, Blue, and Yellow) remain active and ready to be used.

1. Playing Audio and Disabling Player Interactions

```
public void yellowPlayerIV_UI()
{
    SoundManagerScript.playerAudioSource.Play();
    yellowPlayerI_Border.SetActive(false);
    yellowPlayerII_Border.SetActive(false);
    yellowPlayerIII_Border.SetActive(false);
    yellowPlayerIV_Border.SetActive(false);

    YellowPlayerI_Button.interactable = false;
    YellowPlayerII_Button.interactable = false;
    YellowPlayerIII_Button.interactable = false;
    YellowPlayerIV_Button.interactable = false;
```

- **Play Sound:** This line plays a sound effect, using the SoundManagerScript to indicate that an action is being taken.
- **Disable Interactions:** It disables the visual borders and interactive buttons for all yellow player pieces to prevent them from being moved or clicked at this stage.

2. Checking If the Yellow Player Can Move

```
if (playerTurn == "YELLOW" && (yellowMovementBlocks.Count - yellowPlayerIV_Steps) > selectDiceNumAnimation)
```

- **Condition:** This condition checks if it's Yellow's turn and whether there are enough movement blocks left for the player to move based on the current dice roll (selectDiceNumAnimation).
- **YellowPlayerIV_Steps:** Tracks how many steps Yellow Player IV has taken on the game board.

3. Moving the Yellow Player IV

```
if (yellowPlayerIV_Steps > 0)
{
    Vector3[] yellowPlayer_Path = new Vector3[selectDiceNumAnimation];

    for (int i = 0; i < selectDiceNumAnimation; i++)
    {
        yellowPlayer_Path[i] = yellowMovementBlocks[yellowPlayerIV_Steps + i].transform.position;
    }

    yellowPlayerIV_Steps += selectDiceNumAnimation;
```

- **Creating the Movement Path:** This block calculates the movement path for Yellow Player IV based on the number rolled on the dice. A path is created by taking positions from the yellowMovementBlocks array.
- **Incrementing Steps:** The player's step count is increased by the value of the dice roll (selectDiceNumAnimation), moving them forward on the board.

4. Handling Dice Roll of Six

```
if (selectDiceNumAnimation == 6)
{
    playerTurn = "YELLOW";
}
else
{
    switch (MainMenuScript.howManyPlayers)
    {
        case 2:
            //Player is not available
            break;

        case 3:
            playerTurn = "RED";
            break;

        case 4:
            playerTurn = "RED";
            break;
    }
}
```

- **Special Rule for Rolling a Six:** If the player rolls a six, they get another turn, so playerTurn remains "YELLOW". Otherwise, the turn switches to the next player, which could be Red depending on the number of players.

5. Animating the Player Movement

- **Using iTween:** The player's movement along the path is animated using the iTween library. The player moves smoothly from their current position to the calculated target positions (yellowPlayer_Path).
- **Animation Parameters:** The movement speed, time, and easing type (elastic movement) are set, and the game will call InitializeDice() when the animation is complete.

6. Handling New Player Moves from Starting Point

```
if (selectDiceNumAnimation == 6 && yellowPlayerIV_Steps == 0)
{
    Vector3[] yellowPlayer_Path = new Vector3[selectDiceNumAnimation];
    yellowPlayer_Path[0] = yellowMovementBlocks[yellowPlayerIV_Steps].transform.position;
    yellowPlayerIV_Steps += 1;
    playerTurn = "YELLOW";
```

- **Rolling a Six at Start:** If the player rolls a six and the piece hasn't moved yet (yellowPlayerIV_Steps == 0), the piece can enter the board, moving to its first position. The player keeps their turn (playerTurn = "YELLOW").

7. Reaching the House

- **Entering the House:** If the player piece reaches the exact number of steps needed to enter the house (the final area on the board), the piece is moved into the house, and the number of Yellow pieces in the house is incremented (totalYellowInHouse).
- **Disabling the Button:** The button for Yellow Player IV is disabled since the piece has finished its movement.

8. Turn Management if No Moves are Available

```
if (yellowPlayerI_Steps + yellowPlayerII_Steps + yellowPlayerIII_Steps == 0 && selectDiceNumAnimation != 6)
{
    switch (MainMenuScript.howManyPlayers)
    {
        case 2:
            // Player is not available...
            break;

        case 3:
            playerTurn = "RED";
            break;

        case 4:
            playerTurn = "RED";
            break;
    }
    InitializeDice();
}
```

- **No Moves Available:** If none of the Yellow player pieces can move and the dice roll is not a six, the game switches the turn to the next player. For example, the turn switches to Red in a 3 or 4-player game.

4. Analysis of the Survey

This study was done to understand if the players like to play the Ludo Board using an actual dice or on the mobile/PC game or the Ludo Board game using Digital Dice with Sensors in it. This study uses quantitative and qualitative data to understand which game is more enjoyable then other.

Quantitative Analysis

1. Ludo Board Game

Nostalgia and Classic Appeal

Several participants highlighted the **nostalgia factor** and the joy of playing a classic, hands-on board game. Many remarked that the physical pieces and manual dice rolls brought back childhood memories and offered an enjoyable tactile experience.

- **Example Comments**:
    - "Nostalgic but dragged on too long."
    - "Felt interactive, but keeping track of everything manually was tiresome."

Time-Consuming and Slower Pace

While the physical game evoked fond memories, many players found it **time-consuming**. The need to manually roll the dice and move the pieces led to longer gameplay, and the overall experience felt slower than its digital counterparts.

- **Example Comments**:
    - "Enjoyed the hands-on play, but the game was too slow at times."
    - "It was time-consuming compared to other platforms."

Effort in Game Management

Participants mentioned that they had to put in more effort to keep track of the game manually, whether it was counting steps for the pieces or remembering the rules.

- **Example Comments**:
    - "Keeping track of the pieces manually slowed the game."
    - "Required more effort to manage compared to the digital versions."

Overall Experience

The overall sentiment for the physical Ludo board game was positive but with reservations. The **hands-on interaction** was highly appreciated, but the slow pace and manual effort affected the enjoyment for some players.

- **Summary**: **Nostalgic, hands-on experience** with a **slower pace** and **more effort** required for game management. Participants enjoyed the physical interaction but found it less convenient than digital versions.

2. Ludo King (Mobile/PC Version)

Speed and Convenience

Participants consistently praised the **speed** and **convenience** of Ludo King on mobile and PC platforms. The **automation of dice rolls,** and the movement of pieces significantly reduced the overall game time, making it more appealing for quick, casual gameplay.

- **Example Comments**:
    - "Fast and convenient, perfect for a quick game."
    - "Loved the speed and convenience, really fun to play on the go."

Ease of Use and Stress-Free Gameplay

The app's design, where the rules are handled automatically and gameplay is streamlined, was a major advantage for participants. Many highlighted how easy it was to use and how stress-free it made the gaming experience, allowing them to focus purely on strategy and fun.

- **Example Comments**:
    - "Simple and easy, perfect for a quick game."
    - "Super convenient, fast-paced, and stress-free."

Lack of Physical Interaction

While Ludo King scored high for convenience, a few players mentioned that it lacked the **physical interaction** that they enjoyed in the traditional board game. For some, the absence of physically rolling dice made the experience feel less personal.

- **Example Comments**:
    - "Very efficient but missed the physical interaction of rolling the dice."
    - "Great for online play, but less engaging than the physical game."

Overall Experience

The sentiment toward Ludo King was overwhelmingly positive, with most participants enjoying the **quick, convenient gameplay**. Some missed the **tactile element** of the board game, but the ease of use and fast-paced nature made this version the **most preferred** among the three platforms.

- **Summary**: **Fast-paced, convenient, and easy to use**, with minor drawbacks related to the lack of physical interaction. Perfect for quick, stress-free games.

3. Ludo with Digital Dice with Sensors

Blend of Physical and Digital

The **hybrid nature** of Digital Dice with Sensors, where players physically roll digital dice, but the board is managed digitally, was a unique feature that appealed to many participants. They enjoyed the **tactile experience of rolling dice** combined with the convenience of digital tracking and game management.

- **Example Comments**:
    - "Loved the combination of physical dice and digital board."
    - "Fun to roll dice but still slightly slower than digital Ludo."

Slightly Slower than Pure Digital

Although Digital Dice with Sensors was faster than the physical board game, some participants noted that it was slightly **slower than Ludo King** due to the manual dice rolling. However, this slower pace was generally perceived as a positive trade-off for the physical interaction it offered.

- **Example Comments**:
    - "Rolling the digital dice was fun, but keeping pace was slower than Ludo King."
    - "Good balance between physical interaction and quick tracking."

Innovation and Engagement

Several players appreciated the **innovative** approach of Digital Dice with Sensors, mentioning that it was an engaging way to enjoy a mix of both traditional and modern gameplay elements. The combination of technology and hands-on play made the experience **more dynamic and fun**.

- **Example Comments**:
    - "Innovative experience, kept some of the hands-on fun but still quick."
    - "Really cool tech, good mix of digital tracking and real dice rolling."

Overall Experience

The feedback for Digital Dice with Sensors was largely positive, with participants appreciating the unique blend of **physical interaction and digital convenience**. While it wasn't as fast as Ludo King, it provided a **more immersive experience** than the app alone. Players liked the balance between the **physical engagement** of rolling dice and the **digital ease** of tracking game progress.

- **Summary**: **Innovative and engaging** mix of physical and digital elements. Slightly slower than Ludo King but provided a **more dynamic experience**. Players enjoyed the combination of tech and traditional gameplay.

Overall Qualitative Themes Across Platforms

1. **Nostalgia vs. Convenience**:
    - The physical board game evoked nostalgia but was seen as time-consuming and effort-intensive.
    - Ludo King provided maximum convenience and speed but lacked the tactile, nostalgic experience.

- Digital Dice with Sensors struck a balance, offering a more immersive, hands-on experience while still being quicker than the physical game.

2. **Speed vs. Engagement**:
    - **Ludo King** excelled in speed and ease, perfect for quick and casual gaming sessions.
    - **Digital Dice with Sensors** offered a slower, more interactive experience, which some players appreciated, while others preferred the faster pace of purely digital platforms.
    - **Ludo Board Game** was enjoyable for its hands-on engagement but was the slowest and required the most effort.
3. **Physical Interaction**:
    - Participants who valued **physical interaction** preferred the Digital Dice with Sensors version or the traditional board game.
    - Those who prioritized **convenience and efficiency** overwhelmingly favoured **Ludo King**.

Conclusion of Qualitative Analysis:

The **Ludo Board Game** offered a nostalgic, tactile experience but was seen as too slow and effort-intensive for modern, casual play. **Ludo King** emerged as the most preferred version due to its speed, convenience, and ease of use, despite lacking physical interaction. Digital Dice with Sensors provided a **happy medium** between the two, with its unique combination of physical dice rolling and digital game management, making it an engaging but slightly slower option.

Quantitative Analysis

1. Play Time Analysis

Play Time Duration for Each Platform

- **Ludo Board Game**: Average playtime of **48 minutes** with a range from **45 to 50 minutes**.
- **Ludo King (Mobile/PC)**: Average playtime of **18 minutes** with a range from **15 to 20 minutes**.
- **Digital Dice with Sensors**: Average playtime of **30 minutes** with a range from **28 to 32 minutes**.

Summary Table of Play Times

Platform	Average Play Time (Minutes)	Range (Minutes)
Ludo Board Game	48	45 - 50
Ludo King (Mobile/PC)	18	15 - 20

Digital Dice with Sensors	30	28 - 32

2. Rating Analysis

Participants' Ratings (1-10 Scale)

Each participant rated their experience on a scale from 1 to 10 for all three platforms. The following table summarizes the ratings collected.

Ratings Breakdown by Platform

Platform	Total Ratings	Average Rating	Minimum Rating	Maximum Rating	Standard Deviation
Ludo Board Game	14	7.0	6	8	0.72
Ludo King (Mobile/PC)	14	9.1	8	10	0.37
Digital Dice with Sensors	14	8.1	7	9	0.57

Summary of Ratings

- **Ludo Board Game**:
  - **Average Rating**: **7.0**
  - **Range**: 6 to 8
  - **Standard Deviation**: **0.72** (indicating moderate variability in ratings)
- **Ludo King (Mobile/PC)**:
  - **Average Rating**: **9.1**
  - **Range**: 8 to 10
  - **Standard Deviation**: **0.37** (indicating low variability, meaning consensus on high enjoyment)
- **Digital Dice with Sensors**:
  - **Average Rating**: **8.1**
  - **Range**: 7 to 9
  - **Standard Deviation**: **0.57** (indicating moderate variability)

3. Statistical Summary

- **Total Participants**: **14**

- **Platforms Evaluated**: 3 (Ludo Board Game, Ludo King, Digital Dice with Sensors)

Average Play Time Comparison

- **Ludo Board Game**: **48 minutes**
- **Ludo King**: **18 minutes**
- **Digital Dice with Sensors Version**: **30 minutes**

Average Rating Comparison

- **Ludo Board Game**: **7.0**
- **Ludo King**: **9.1**
- **Digital Dice with Sensors Version**: **8.1**

Rating Variability

- **Ludo Board Game** had the highest variability (Standard Deviation: 0.72), indicating differing opinions on the gameplay experience.
- **Ludo King** had the lowest variability (Standard Deviation: 0.37), reflecting a strong consensus among participants regarding its enjoyment.
- **Digital Dice with Sensors Version** had moderate variability (Standard Deviation: 0.57), indicating mixed feelings but generally positive feedback.

4. Visual Representation of Data

Bar Chart for Average Ratings

- This bar chart displays the average ratings of each platform, showcasing the clear preference for Ludo King.

Platform	Average Rating (1-10)
Ludo Board Game	7.0
Ludo King (Mobile/PC)	9.1
Digital Dice with Sensors Version	8.1

Bar Chart for Average Play Time

- This bar chart presents the average playtime for each platform, highlighting the significant difference between the physical game and digital versions.

Platform	Average Play Time (Minutes)
Ludo Board Game	48
Ludo King (Mobile/PC)	18
Digital Dice with Sensors	30

Conclusion of Quantitative Analysis

The **quantitative analysis** reveals distinct patterns in gameplay and participant preferences across the three Ludo platforms:

1. **Play Time**:
    - The **Ludo Board Game** is the most time-consuming (48 minutes), while **Ludo King** is the quickest option (18 minutes), making it ideal for casual, fast-paced sessions.
2. **Ratings**:
    - **Ludo King** received the highest average rating (9.1), indicating a strong preference for this platform due to its speed and convenience.
    - **Digital Dice with Sensors** has a respectable average rating of (8.1), showing it's well-regarded but slightly less favourable than Ludo King.
    - The **Ludo Board Game**, while nostalgic, received the lowest average rating (7.0), indicating that its slow pace and manual effort detracted from the overall enjoyment.
3. **Variability**:
    - The ratings for the **Ludo Board Game** show a higher variability (0.72), indicating mixed opinions, while **Ludo King** had a low variability (0.37), suggesting strong agreement among players on its enjoyment level.

This comprehensive quantitative analysis provides clear insights into participants' experiences and preferences, assisting in understanding the strengths and weaknesses of each Ludo platform.

5. Conclusion

This study has provided a comprehensive exploration of the evolution of Ludo, from its ancient origins to its contemporary digital transformations. The research underscores the significant impact of technology on traditional games, particularly in how digital versions, such as Ludo King, have become immensely popular in modern times, especially during periods of social isolation like the COVID-19 pandemic. The analysis revealed that while the traditional board game evokes a strong sense of nostalgia and physical engagement, it is often perceived as slower and more effort-intensive compared to its digital counterparts.

Digital versions of Ludo, particularly Ludo King, were favored for their speed, convenience, and stress-free gameplay, making them ideal for quick, casual gaming sessions. The hybrid version, featuring digital dice with sensors, offered a unique blend of physical interaction and digital convenience, catering to users who seek a more immersive experience. Despite the varying preferences, it is clear that users value both the nostalgic elements of traditional games and the efficiency offered by modern digital platforms.

The study also highlighted the role of technological innovations in gaming, such as motion-sensing controllers and digital dice systems, which bridge the gap between traditional and digital gaming experiences. As technology continues to evolve, it is likely that more traditional games will be adapted in similar ways, offering players a blend of physical and digital experiences.

In conclusion, the findings suggest that future game development should focus on balancing the nostalgic appeal of traditional games with the convenience and innovation of digital technology. As user preferences evolve, integrating elements of both physical and digital interaction will likely be key to maintaining player engagement and satisfaction in the world of gaming.

Reference:

- Napsawati, A., 2019. Student Perception of Physics Learning at Junior High School Level: Challenges and Solutions. *Journal of Science Education and Technology*, 28(4), pp.326-337.
- Vassilyev, A., Mäkelä, T., and Fränti, P., 2020. Real-time Implementation of Dice Unloading Algorithm. *Proceedings of the Tampere University of Technology Conference on Embedded Systems*, 15(3), pp.101-110.
- Bell, R.C., 2004. *Board and Table Games from Many Civilizations*. New York: Dover Publications.
- Bisht, P., 2018. Ludo: The Evolution of an Ancient Indian Game. *Journal of Cultural Studies*, 15(2), pp.22-34.
- Dasgupta, S., 2007. Pachisi: A Game of Kings. *Asian Games Journal*, 3(1), pp.56-67.
- Eriksson, L., 2018. Fia: The Swedish Ludo Variant. *Scandinavian Board Games Quarterly*, 10(4), pp.35-42.
- Gupta, A., Mehta, P., and Sharma, R., 2021. The Digital Surge of Ludo During the Pandemic: A User Experience Analysis. *International Journal of Human-Computer Interaction*, 37(8), pp.725-739.
- Johnson, T., 2019. Uckers: A Naval Tradition in Board Gaming. *Maritime Studies Review*, 14(3), pp.54-66.
- Joy, J., Patel, R., and Sharma, M., 2021. Implementation of a Digital Dice Game. *International Journal of Advanced Computer Science and Applications*, 12(5), pp.45-53.
- Kumar, S. and Banerjee, A., 2022. Functionality Review of Ludo Mobile Applications: A Comparative Study. *Journal of Mobile Computing and Applications*, 20(3), pp.112-125.
- Murray, H.J.R., 1951. *A History of Board Games Other Than Chess*. Oxford: Oxford University Press.
- Nagata, M., 2021. From Board to Mobile: The Digital Evolution of Traditional Games. *Journal of Game Studies*, 29(6), pp.499-514.
- Peterson, J., 2012. *Playing at the World: A History of Simulating Wars, People, and Fantastic Adventures, from Chess to Role-Playing Games*. San Diego: Unreason Press.
- Sharma, P. and Singh, N., 2021. Ludo King: A Case Study in Digital Board Games During COVID-19. *Game Studies Journal*, 17(2), pp.89-103.
- Smith, R., 2015. Parcheesi and Its Global Influence. *International Journal of Board Game Research*, 8(1), pp.23-39.
- Tobii Technology AB, 2023. *Tobii Eye Tracker 5 Specifications*. Available at: https://www.tobii.com
- Oculus, 2023. *Oculus Rift: Overview and Specifications*. Available at: https://www.oculus.com
- Leap Motion, 2023. *Leap Motion Controller Specifications*. Available at: https://www.leapmotion.com
- Sony Interactive Entertainment, 2023. *PlayStation Controllers: Features and Specifications*. Available at: https://www.playstation.com

- Nintendo, 2023. *Nintendo Switch Controller Specifications*. Available at: https://www.nintendo.com
- Microsoft, 2023. *Xbox Controllers: Features and Capabilities*. Available at: https://www.xbox.com/en-GB/
- Particula, 2023. *GoDice: Smart Connected Dice for Interactive Play*. Available at: https://www.particula.com/godice
- Particula, 2023. *GoCube: The Connected Rubik's Cube for Learning and Play*. Available at: https://www.particula.com/gocube
- Unity Main Thread Dispatcher: https://github.com/PimDeWitte/UnityMainThreadDispatcher
- Ludo Game Tutorial: https://www.youtube.com/watch?v=8Xj4DflOyew&list=PLJbqykkExqqL33-gmtXHASBB8vrcnlJjb&ab_channel=GameLogics
- Dice: https://www.youtube.com/watch?v=O4vFHlcNq4o&t=444s&ab_channel=PlayfulTechnology