ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

ΠΡΟΓΡΑΜΜΑ ΔΙΔΑΚΤΟΡΙΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Περιβάλλοντα Επαυξημένης/Εικονικής Πραγματικότητας για Δοκιμασίες στο πεδίο της Εκπαίδευσης: πρακτική εφαρμογή και αξιολόγηση

Βαΐα Μαραγκού

ΑΙΓΑΛΕΩ

ΜΑΡΤΙΟΣ 2026

ΠΑΡΑΡΤΗΜΑ Γ

DESKTOP VR – VRBCS 1

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 1

MoveCubeAround

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class MoveCubeAround : MonoBehaviour
{
    // Directional buttons
    public Button forwardButton;
    public Button rightButton;
    public Button backwardButton;
    public Button leftButton;

    // Magnitude buttons
    public Button magnitudeButton1;
    public Button magnitudeButton2;
    public Button magnitudeButton3;
    public Button magnitudeButton4;
    public Button magnitudeButton5;
    public Button magnitudeButton6;
    public Button magnitudeButton7;
    public Button magnitudeButton8;

    // Run button
    public Button runButton;
```

```
    public MoveAround targetScript;  // Script that contains the movement logic

    public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI component

    private bool readyToMoveForward = false;
    private bool readyToMoveRight = false;
    private bool readyToMoveBackward = false;
    private bool readyToMoveLeft = false;
    private int moveMagnitude = 0;  // To store the movement magnitude

    void Start()
    {
        // Assign button click listeners for directions
        forwardButton.onClick.AddListener(() => SetDirection("forward"));
        rightButton.onClick.AddListener(() => SetDirection("right"));
        backwardButton.onClick.AddListener(() => SetDirection("backward"));
        leftButton.onClick.AddListener(() => SetDirection("left"));

        // Assign button click listeners for magnitudes
        magnitudeButton1.onClick.AddListener(() => SetMagnitude(1));
        magnitudeButton2.onClick.AddListener(() => SetMagnitude(2));
        magnitudeButton3.onClick.AddListener(() => SetMagnitude(3));
        magnitudeButton4.onClick.AddListener(() => SetMagnitude(4));
        magnitudeButton5.onClick.AddListener(() => SetMagnitude(5));
        magnitudeButton6.onClick.AddListener(() => SetMagnitude(6));
        magnitudeButton7.onClick.AddListener(() => SetMagnitude(7));
        magnitudeButton8.onClick.AddListener(() => SetMagnitude(8));

        // Assign button click listener for the run action
        runButton.onClick.AddListener(ProcessMove);
```

```
}

private void SetDirection(string direction)
{
    ResetReadyStates();
    switch (direction)
    {
        case "forward":
            readyToMoveForward = true;
            break;
        case "right":
            readyToMoveRight = true;
            break;
        case "backward":
            readyToMoveBackward = true;
            break;
        case "left":
            readyToMoveLeft = true;
            break;
    }
}

private void SetMagnitude(int magnitude)
{
    moveMagnitude = magnitude;
}

private void ProcessMove()
{
    if (moveMagnitude > 0)
    {
```

```
            string message = "";
            if (readyToMoveForward)
            {
                targetScript.MoveCube("forward", moveMagnitude);
                message = $"moveForward({moveMagnitude})";
            }
            else if (readyToMoveRight)
            {
                targetScript.MoveCube("right", moveMagnitude);
                message = $"moveRight({moveMagnitude})";
            }
            else if (readyToMoveBackward)
            {
                targetScript.MoveCube("backward", moveMagnitude);
                message = $"moveBackward({moveMagnitude})";
            }
            else if (readyToMoveLeft)
            {
                targetScript.MoveCube("left", moveMagnitude);
                message = $"moveL
eft({moveMagnitude})";
            }

            actionText.text = message;  // Update the UI text
        }
        ResetReadyStates(); // Reset the states after processing the move
    }

    private void ResetReadyStates()
    {
        readyToMoveForward = false;
```

```
        readyToMoveRight = false;
        readyToMoveBackward = false;
        readyToMoveLeft = false;
        moveMagnitude = 0;  // Reset magnitude for next input
    }
}
```

MoveAround

```
using UnityEngine;


public class MoveAround : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector


   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };


   void Start()
   {
      // Set the cube's starting position
      cube.transform.position = new Vector3(10f, 0.08f, 10f);
      Debug.Log("Cube Start Position: " + cube.transform.position);
   }


   public void MoveCube(string direction, int magnitude)
   {
      Vector3 proposedMove = Vector3.zero;
```

```
        switch (direction)
        {
            case "forward":
                proposedMove = Vector3.right * magnitude; // Moves in +X direction
                break;
            case "backward":
                proposedMove = Vector3.left * magnitude; // Moves in -X direction
                break;
            case "right":
                proposedMove = Vector3.back * magnitude; // Moves in -Z direction
                break;
            case "left":
                proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
                break;
        }

        Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");

        if (CanMove(proposedMove))
        {
            cube.transform.position += proposedMove;
            Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
        }
        else
        {
            Debug.Log($"Move blocked: {direction} at position {cube.transform.position +
proposedMove}");
        }
    }
```

```
    private bool CanMove(Vector3 proposedMove)
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point
precision
            {
                return true;
            }
        }
        return false; // Block move if it does not exactly match any allowed position
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 2

MoveCubeAround2

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class MoveCubeAround2 : MonoBehaviour
{
    // Directional buttons
    public Button forwardButton;
    public Button rightButton;
    public Button backwardButton;
    public Button leftButton;

    // Magnitude buttons
    public Button magnitudeButton1;
    public Button magnitudeButton2;
    public Button magnitudeButton3;
    public Button magnitudeButton4;
    public Button magnitudeButton5;
    public Button magnitudeButton6;
    public Button magnitudeButton7;
    public Button magnitudeButton8;

    // Run button
    public Button runButton;

    public MoveAround2 targetScript;  // Script that contains the movement logic

    public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component
```

```
private bool readyToMoveForward = false;
private bool readyToMoveRight = false;
private bool readyToMoveBackward = false;
private bool readyToMoveLeft = false;
private int moveMagnitude = 0;  // To store the movement magnitude

void Start()
{
    // Assign button click listeners for directions
    forwardButton.onClick.AddListener(() => SetDirection("forward"));
    rightButton.onClick.AddListener(() => SetDirection("right"));
    backwardButton.onClick.AddListener(() => SetDirection("backward"));
    leftButton.onClick.AddListener(() => SetDirection("left"));

    // Assign button click listeners for magnitudes
    magnitudeButton1.onClick.AddListener(() => SetMagnitude(1));
    magnitudeButton2.onClick.AddListener(() => SetMagnitude(2));
    magnitudeButton3.onClick.AddListener(() => SetMagnitude(3));
    magnitudeButton4.onClick.AddListener(() => SetMagnitude(4));
    magnitudeButton5.onClick.AddListener(() => SetMagnitude(5));
    magnitudeButton6.onClick.AddListener(() => SetMagnitude(6));
    magnitudeButton7.onClick.AddListener(() => SetMagnitude(7));
    magnitudeButton8.onClick.AddListener(() => SetMagnitude(8));

    // Assign button click listener for the run action
    runButton.onClick.AddListener(ProcessMove);
}

private void SetDirection(string direction)
{
```

```
    ResetReadyStates();
    switch (direction)
    {
        case "forward":
            readyToMoveForward = true;
            break;
        case "right":
            readyToMoveRight = true;
            break;
        case "backward":
            readyToMoveBackward = true;
            break;
        case "left":
            readyToMoveLeft = true;
            break;
    }
}

private void SetMagnitude(int magnitude)
{
    moveMagnitude = magnitude;
}

private void ProcessMove()
{
    if (moveMagnitude > 0)
    {
        string message = "";
        if (readyToMoveForward)
        {
            targetScript.MoveCube("forward", moveMagnitude);
```

```
            message = $"moveForward({moveMagnitude})";
        }
        else if (readyToMoveRight)
        {
            targetScript.MoveCube("right", moveMagnitude);
            message = $"moveRight({moveMagnitude})";
        }
        else if (readyToMoveBackward)
        {
            targetScript.MoveCube("backward", moveMagnitude);
            message = $"moveBackward({moveMagnitude})";
        }
        else if (readyToMoveLeft)
        {
            targetScript.MoveCube("left", moveMagnitude);
            message = $"moveLeft({moveMagnitude})";
        }

        actionText.text = message;  // Update the UI text
    }
    ResetReadyStates(); // Reset the states after processing the move
}

private void ResetReadyStates()
{
    readyToMoveForward = false;
    readyToMoveRight = false;
    readyToMoveBackward = false;
    readyToMoveLeft = false;
    moveMagnitude = 0;  // Reset magnitude for next input
}
```

```
}
```

MoveAround2

```
using UnityEngine;


public class MoveAround2 : MonoBehaviour
{
    public GameObject cube; // Assign your Cube GameObject in the inspector


    private Vector3[] allowedPositions = new Vector3[]
    {
        new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f, 10),
        new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f, 10),
        new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
        new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f, 12),
        new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
        new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
    };


    void Start()
    {
        // Set the cube's starting position and initial rotation
        cube.transform.position = new Vector3(10f, 0.08f, 10f);
        cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Sets initial rotation to 0, 90, 0
        Debug.Log("Cube Start Position: " + cube.transform.position);
    }


    public void MoveCube(string direction, int magnitude)
```

```
  {
      Vector3 proposedMove = Vector3.zero;


      switch (direction)
      {
          case "forward":
              proposedMove = Vector3.right * magnitude; // Moves in +X direction
              cube.transform.rotation = Quaternion.Euler(0, 90, 0); // Set rotation for
moving forward
              break;
          case "backward":
              proposedMove = Vector3.left * magnitude; // Moves in -X direction
              cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Set rotation for
moving backward
              break;
          case "right":
              proposedMove = Vector3.back * magnitude; // Moves in -Z direction
              cube.transform.rotation = Quaternion.Euler(0, 180, 0); // Set rotation for
moving right
              break;
          case "left":
              proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
              cube.transform.rotation = Quaternion.Euler(0, 0, 0); // Set rotation for
moving left (or you can use 360, both are the same)
              break;
      }


      Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");


      if (CanMove(proposedMove))
      {
```

```
            cube.transform.position += proposedMove;
            Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
        }
        else
        {
            Debug.Log($"Move blocked: {direction} at position {cube.transform.position + proposedMove}");
        }
    }

    private bool CanMove(Vector3 proposedMove)
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point precision
            {
                return true;
            }
        }
        return false; // Block move if it does not exactly match any allowed position
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 3

MoveAround2_CHANGED

```
using UnityEngine;


public class MoveAround2_CHANGED : MonoBehaviour


{
   public GameObject cube; // Assign your Cube GameObject in the inspector


   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };


   void Start()
   {
      // Set the cube's starting position and initial rotation
      cube.transform.position = new Vector3(10f, 0.08f, 10f);
      cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Sets initial rotation to
0, 90, 0
      Debug.Log("Cube Start Position: " + cube.transform.position);
   }


   public void MoveCube(string direction, int magnitude)
```

```
  {
      Vector3 proposedMove = Vector3.zero;


      switch (direction)
      {
          case "forward":
              proposedMove = Vector3.right * magnitude; // Moves in +X direction
              cube.transform.rotation = Quaternion.Euler(0, 90, 0); // Set rotation for
moving forward
              break;
          case "backward":
              proposedMove = Vector3.left * magnitude; // Moves in -X direction
              cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Set rotation for
moving backward
              break;
          case "right":
              proposedMove = Vector3.back * magnitude; // Moves in -Z direction
              cube.transform.rotation = Quaternion.Euler(0, 180, 0); // Set rotation for
moving right
              break;
          case "left":
              proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
              cube.transform.rotation = Quaternion.Euler(0, 0, 0); // Set rotation for
moving left (or you can use 360, both are the same)
              break;
      }


      Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");


      if (CanMove(proposedMove))
      {
```

```
            cube.transform.position += proposedMove;
            Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
        }
        else
        {
            Debug.Log($"Move blocked: {direction} at position {cube.transform.position + proposedMove}");
        }
    }

    private bool CanMove(Vector3 proposedMove)
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point precision
            {
                return true;
            }
        }
        return false; // Block move if it does not exactly match any allowed position
    }
}
```

MoveCubeAround2_CHANGED

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class MoveCubeAround2_CHANGED : MonoBehaviour
{
    // Directional buttons
    public Button forwardButton;
    public Button rightButton;
    public Button backwardButton;
    public Button leftButton;

    // Magnitude buttons
    public Button magnitudeButton1;
    public Button magnitudeButton2;
    public Button magnitudeButton3;
    public Button magnitudeButton4;
    public Button magnitudeButton5;
    public Button magnitudeButton6;
    public Button magnitudeButton7;
    public Button magnitudeButton8;

    // Run button
    public Button runButton;

    // Reference to the script containing the movement logic
    public MoveAround2_CHANGED targetScript;

    // TextMeshPro UI component to display the action
```

```
public TextMeshProUGUI actionText;

private bool readyToMoveForward = false;
private bool readyToMoveRight = false;
private bool readyToMoveBackward = false;
private bool readyToMoveLeft = false;
private int moveMagnitude = 0;  // To store the movement magnitude

void Start()
{
    // Assign button click listeners for directions
    forwardButton.onClick.AddListener(() => SetDirection("forward"));
    rightButton.onClick.AddListener(() => SetDirection("right"));
    backwardButton.onClick.AddListener(() => SetDirection("backward"));
    leftButton.onClick.AddListener(() => SetDirection("left"));

    // Assign button click listeners for magnitudes
    magnitudeButton1.onClick.AddListener(() => SetMagnitude(1));
    magnitudeButton2.onClick.AddListener(() => SetMagnitude(2));
    magnitudeButton3.onClick.AddListener(() => SetMagnitude(3));
    magnitudeButton4.onClick.AddListener(() => SetMagnitude(4));
    magnitudeButton5.onClick.AddListener(() => SetMagnitude(5));
    magnitudeButton6.onClick.AddListener(() => SetMagnitude(6));
    magnitudeButton7.onClick.AddListener(() => SetMagnitude(7));
    magnitudeButton8.onClick.AddListener(() => SetMagnitude(8));

    // Assign button click listener for the run action
    runButton.onClick.AddListener(ProcessMove);
}

private void SetDirection(string direction)
```

```
{
    ResetReadyStates();
    switch (direction)
    {
        case "forward":
            readyToMoveForward = true;
            break;
        case "right":
            readyToMoveRight = true;
            break;
        case "backward":
            readyToMoveBackward = true;
            break;
        case "left":
            readyToMoveLeft = true;
            break;
    }
}

private void SetMagnitude(int magnitude)
{
    moveMagnitude = magnitude;
}

private void ProcessMove()
{
    if (moveMagnitude > 0)
    {
        string direction = "";
        if (readyToMoveForward)
        {
```

```
            targetScript.MoveCube("forward", moveMagnitude);
            direction = "Forward";
        }
        else if (readyToMoveRight)
        {
            targetScript.MoveCube("right", moveMagnitude);
            direction = "Right";
        }
        else if (readyToMoveBackward)
        {
            targetScript.MoveCube("backward", moveMagnitude);
            direction = "Backward";
        }
        else if (readyToMoveLeft)
        {
            targetScript.MoveCube("left", moveMagnitude);
            direction = "Left";
        }

        // Update the UI text to the new format
        actionText.text = $"move ({direction}, {moveMagnitude})";
    }
    ResetReadyStates(); // Reset the states after processing the move
}

private void ResetReadyStates()
{
    readyToMoveForward = false;
    readyToMoveRight = false;
    readyToMoveBackward = false;
    readyToMoveLeft = false;
```

```
        moveMagnitude = 0;  // Reset magnitude for next input
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 4

MoveCubeAround7

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;


public class MoveCubeAround7 : MonoBehaviour
{
    public Button West, North, East, South, NW, NE, SE, SW;
    public Button[] magnitudeButtons;
    public Button runButton;
    public MoveAround7 targetScript;  // Reference to the script that contains the movement logic
    public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI component


    private string currentDirection = "";
    private int moveMagnitude = 0;
    private string displayText = "";


    void Start()
    {
        West.onClick.AddListener(() => SetDirection("forward", "MoveWest"));
        North.onClick.AddListener(() => SetDirection("right", "MoveNorth"));
        East.onClick.AddListener(() => SetDirection("backward", "MoveEast"));
        South.onClick.AddListener(() => SetDirection("left", "MoveSouth"));
        NW.onClick.AddListener(() => SetDirection("NE", "MoveNW"));
        NE.onClick.AddListener(() => SetDirection("SE", "MoveNE"));
        SE.onClick.AddListener(() => SetDirection("SW", "MoveSE"));
        SW.onClick.AddListener(() => SetDirection("NW", "MoveSW"));
```

```
        foreach (var button in magnitudeButtons)
        {
            button.onClick.AddListener(() =>
            {
SetMagnitude(int.Parse(button.GetComponentInChildren<TextMeshProUGUI>().text)
);
            });
        }

        runButton.onClick.AddListener(ProcessMove);
    }

    private void SetDirection(string direction, string displayText)
    {
        currentDirection = direction;
        this.displayText = displayText;
        Debug.Log($"Direction set to {currentDirection}");
    }

    private void SetMagnitude(int magnitude)
    {
        moveMagnitude = magnitude;
        Debug.Log($"Magnitude set to {moveMagnitude}");
    }

    private void ProcessMove()
    {
        if (moveMagnitude > 0 && !string.IsNullOrEmpty(currentDirection))
        {
            Vector3 initialPosition = targetScript.cube.transform.position;
```

```
        Vector3 targetPosition = CalculateTargetPosition(currentDirection,
moveMagnitude);
        targetScript.MoveCube(targetPosition);

        if (targetScript.cube.transform.position != initialPosition)
        {
            UpdateActionText($"{displayText} ({moveMagnitude})");
        }
        else
        {
            UpdateActionText($"{displayText} ({moveMagnitude})");
        }
    }
    else
    {
        UpdateActionText("Move command ignored: No direction or magnitude set.");
    }
}

private Vector3 CalculateTargetPosition(string direction, int magnitude)
{
    Vector3 currentPosition = targetScript.cube.transform.position;
    switch (direction)
    {
        case "forward":
            return currentPosition + new Vector3(0, 0, magnitude);
        case "right":
            return currentPosition + new Vector3(magnitude, 0, 0);
        case "backward":
            return currentPosition - new Vector3(0, 0, magnitude);
        case "left":
```

```
            return currentPosition - new Vector3(magnitude, 0, 0);
        case "NE":
            return currentPosition + new Vector3(magnitude, 0, magnitude);
        case "SE":
            return currentPosition + new Vector3(magnitude, 0, -magnitude);
        case "SW":
            return currentPosition - new Vector3(magnitude, 0, magnitude);
        case "NW":
            return currentPosition - new Vector3(magnitude, 0, -magnitude);
        default:
            return currentPosition; // Return current position if the direction is invalid
    }
  }

  private void UpdateActionText(string text)
  {
    actionText.text = text;
    Debug.Log($"Action text updated to: {actionText.text}");
  }
}
```

MoveAround7

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


public class MoveAround7 : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector


   // Array of allowed positions for the cube to move to
   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(10, 0.08f, 10), // Start position
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 12), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f, 14), // N1, N2, N3, N4
      new Vector3(11, 0.08f, 10), new Vector3(12, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(14, 0.08f, 10), // E1, E2, E3, E4
      new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8), new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6), // S1, S2, S3, S4
      new Vector3(9, 0.08f, 10), new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10), // W1, W2, W3, W4
      new Vector3(11, 0.08f, 11), new Vector3(12, 0.08f, 12), new Vector3(13, 0.08f, 13), new Vector3(14, 0.08f, 14), // NE1, NE2, NE3, NE4
      new Vector3(11, 0.08f, 9), new Vector3(12, 0.08f, 8), new Vector3(13, 0.08f, 7), new Vector3(14, 0.08f, 6), // SE1, SE2, SE3, SE4
      new Vector3(9, 0.08f, 9), new Vector3(8, 0.08f, 8), new Vector3(7, 0.08f, 7), new Vector3(6, 0.08f, 6), // SW1, SW2, SW3, SW4
      new Vector3(9, 0.08f, 11), new Vector3(8, 0.08f, 12), new Vector3(7, 0.08f, 13), new Vector3(6, 0.08f, 14) // NW1, NW2, NW3, NW4
   };


   void Start()
   {
```

```
        cube.transform.position = new Vector3(10f, 0.08f, 10f); // Set the cube's starting position
        cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Set the cube's starting rotation
    }

    public void MoveCube(Vector3 newPosition)
    {
        if (CanMove(newPosition))
        {
            Vector3 direction = newPosition - cube.transform.position;
            Quaternion targetRotation = Quaternion.identity;

            // Calculate difference in x and z directions
            float deltaX = direction.x;
            float deltaZ = direction.z;

            if (Mathf.Abs(deltaZ) > Mathf.Abs(deltaX))
            {
                // Moving north or south
                if (deltaZ > 0)
                    targetRotation = Quaternion.Euler(0, 0, 0); // North
                else
                    targetRotation = Quaternion.Euler(0, 180, 0); // South
            }
            else if (Mathf.Abs(deltaX) > Mathf.Abs(deltaZ))
            {
                // Moving east or west
                if (deltaX > 0)
                    targetRotation = Quaternion.Euler(0, 90, 0); // East
                else
```

```
                targetRotation = Quaternion.Euler(0, 270, 0); // West
            }
            else
            {
                // Diagonal movements
                if (deltaX > 0 && deltaZ > 0)
                    targetRotation = Quaternion.Euler(0, 45, 0); // NE
                else if (deltaX > 0 && deltaZ < 0)
                    targetRotation = Quaternion.Euler(0, 135, 0); // SE
                else if (deltaX < 0 && deltaZ < 0)
                    targetRotation = Quaternion.Euler(0, 225, 0); // SW
                else if (deltaX < 0 && deltaZ > 0)
                    targetRotation = Quaternion.Euler(0, 315, 0); // NW
            }

            StartCoroutine(RotateAndMove(cube, newPosition, targetRotation));
        }
        else
        {
            Debug.Log($"Move blocked at position {newPosition}");
        }
    }

    private bool CanMove(Vector3 newPosition)
    {
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Allow some small tolerance
for position matching
            {
                return true;
```

```
        }
    }
    return false;
}

private IEnumerator RotateAndMove(GameObject cube, Vector3 targetPosition, Quaternion targetRotation)
{
    float rotationDuration = 0.5f; // Time to rotate
    float moveDuration = 0.5f; // Time to move
    float elapsedTime = 0.0f;
    Quaternion startRotation = cube.transform.rotation;
    Vector3 startPosition = cube.transform.position;

    // Rotate
    while (elapsedTime < rotationDuration)
    {
        cube.transform.rotation = Quaternion.Lerp(startRotation, targetRotation, elapsedTime / rotationDuration);
        elapsedTime += Time.deltaTime;
        yield return null;
    }
    cube.transform.rotation = targetRotation; // Ensure final rotation

    // Reset elapsed time for movement
    elapsedTime = 0.0f;

    // Move
    while (elapsedTime < moveDuration)
    {
        cube.transform.position = Vector3.Lerp(startPosition, targetPosition, elapsedTime / moveDuration);
```

```
            elapsedTime += Time.deltaTime;
            yield return null;
        }
        cube.transform.position = targetPosition; // Ensure final position
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 5

CubeController

```
using UnityEngine;


public class CubeController : MonoBehaviour
{
   void Start()
   {
      // Ensure the cube starts at position (0,0,0)
      transform.position = new Vector3(0, 0, 0);
   }


   // Method to move the cube to a new position
   public void MoveCube(int x, int z)
   {
      // Set the new position of the cube
      transform.position = new Vector3(x, 0, z);
   }
}
```

ButtonManager

```
using UnityEngine;
using TMPro;

public class ButtonManager : MonoBehaviour
{
   public CubeController cubeController; // Reference to the CubeController script
   public TextMeshProUGUI moveText; // Reference to the Text UI element
   private int? xSelection = null; // Store the x-coordinate selection
   private int? zSelection = null; // Store the z-coordinate selection

   // Method to be called by button press, receiving the button number
   public void OnButtonPress(int buttonNumber)
   {
      Debug.Log($"Button pressed: {buttonNumber}"); // Log which button was pressed

      if (buttonNumber < 8) // First 8 buttons are for x axis
      {
         xSelection = buttonNumber;
         Debug.Log($"X selected: {xSelection}"); // Log x-selection
      }
      else if (buttonNumber >= 8 && buttonNumber < 16) // Second 8 buttons are for z axis
      {
         zSelection = buttonNumber - 8; // Normalize button number to 0-7 for z axis
         Debug.Log($"Z selected: {zSelection}"); // Log z-selection
      }

      // Check if both x and z selections are set
      if (xSelection.HasValue && zSelection.HasValue)
```

```
        {
            Debug.Log($"Executing Move to X={xSelection.Value}, Z={zSelection.Value}");
// Log the move execution
            cubeController.MoveCube(xSelection.Value, zSelection.Value); // Move the
cube to the new position
            moveText.text = $"Move({xSelection}, {zSelection})"; // Update the UI text to
show the move
            xSelection = null; // Reset the x selection for the next move
            zSelection = null; // Reset the z selection for the next move
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 6

LightToggle

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

public class LightToggle : MonoBehaviour
{
    public Light greenLight;
    public Light redLight;
    public TextMeshProUGUI countdownText;
    public float delay = 10f;  // Set this delay time for how long each light should stay on.

    private bool isGreenLightActive;
    private int currentCountdown;

    private void Start()
    {
        StartCoroutine(ToggleLights());
    }

    private IEnumerator ToggleLights()
    {
        while (true)
        {
            // Activate green light and deactivate red light
            greenLight.enabled = true;
            redLight.enabled = false;
            isGreenLightActive = true;
```

```
            yield return StartCoroutine(Countdown());

            // Activate red light and deactivate green light
            greenLight.enabled = false;
            redLight.enabled = true;
            isGreenLightActive = false;
            yield return StartCoroutine(Countdown());
        }
    }

    private IEnumerator Countdown()
    {
        for (int i = (int)delay; i > 0; i--)
        {
            countdownText.text = i.ToString();
            currentCountdown = i;
            yield return new WaitForSeconds(1);
        }
    }

    public bool IsGreenLight()
    {
        return isGreenLightActive;
    }

    public int GetCurrentCountdown()
    {
        return currentCountdown;
    }
}
```

PinocchioMovement

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

public class PinocchioMovement : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(19, 2, 88);
    private Vector3 endPosition = new Vector3(-19, 2, 88);
    private bool isMoving = false;
    private float startTime;
    private float speed;

    public LightToggle lightToggle; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition
    public TextMeshProUGUI scoreText; // Assign this in the inspector

    void Start()
    {
        transform.position = startPosition;
        scoreText.text = "Score: 0/10"; // Initialize the score text at the start
        scoreText.gameObject.SetActive(true); // Show the score text at start
    }

    void Update()
    {
        if (isMoving)
```

```
        {
            float timeSinceStarted = Time.time - startTime;
            float fractionOfJourney = timeSinceStarted / speed;
            transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

            // Check if Pinocchio has reached the end position
            if (Vector3.Distance(transform.position, endPosition) < 0.01f)
            {
                isMoving = false;
            }
        }
    }

    public void StartWalking()
    {
        StartMoving(walkDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() >= 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
        }
        else
        {
```

```
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    public void StartRunning()
    {
        StartMoving(runDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() < 5)
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
            else if (lightToggle.GetCurrentCountdown() >= 5 &&
lightToggle.GetCurrentCountdown() < 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            // Modified condition to update the score to 0/10 when the counter is
between 15 and 10
            else if (lightToggle.GetCurrentCountdown() >= 10 &&
lightToggle.GetCurrentCountdown() <= 15)
            {
                UpdateScore(false); // Update score to 0/10
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
```

```
    }
    else
    {
        UpdateScore(false);
        StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
    }
}

private void StartMoving(float duration)
{
    if (!isMoving)
    {
        isMoving = true;
        startTime = Time.time;
        speed = 1.0f / duration;
    }
}

private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
{
    yield return new WaitForSeconds(delay);
    audioSource.PlayOneShot(clip);
}

private void UpdateScore(bool conditionMet)
{
    if (conditionMet)
        scoreText.text = "Score: 10/10";
    else
        scoreText.text = "Score: 0/10";
}
```

```
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 7

LightToggle_204_2

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

public class LightToggle_204_2 : MonoBehaviour
{
    public Light greenLight;
    public Light redLight;
    public TextMeshProUGUI countdownText;
    public float delay = 10f;  // Set this delay time for how long each light should stay on.

    private bool isGreenLightActive;
    private int currentCountdown;

    private void Start()
    {
        StartCoroutine(ToggleLights());
    }

    private IEnumerator ToggleLights()
    {
        while (true)
        {
            // Activate green light and deactivate red light
            greenLight.enabled = true;
            redLight.enabled = false;
            isGreenLightActive = true;
```

```
            yield return StartCoroutine(Countdown());

            // Activate red light and deactivate green light
            greenLight.enabled = false;
            redLight.enabled = true;
            isGreenLightActive = false;
            yield return StartCoroutine(Countdown());
        }
    }

    private IEnumerator Countdown()
    {
        for (int i = (int)delay; i > 0; i--)
        {
            countdownText.text = i.ToString();
            currentCountdown = i;
            yield return new WaitForSeconds(1);
        }
    }

    public bool IsGreenLight()
    {
        return isGreenLightActive;
    }

    public int GetCurrentCountdown()
    {
        return currentCountdown;
    }
}
```

PinocchioMovement_204_3

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204_3 : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(-144, 2, 88);
    private Vector3 endPosition = new Vector3(-183, 2, 88);
    private bool isMoving = false;
    private float startTime;
    private float duration;

    public LightToggle_204_3 lightToggle_204_3; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition

    void Start()
    {
        Debug.Log("PinocchioMovement_204_3 initialized.");
    }

    void Update()
    {
        if (isMoving)
        {
            MovePinocchio();
        }
    }
```

```
public void StartWalking()
{
    if (Vector3.Distance(transform.position, startPosition) < 0.5f)
    {
        StartMoving(walkDuration);
        EvaluateConditions(isWalking: true);
    }
}

public void StartRunning()
{
    if (Vector3.Distance(transform.position, startPosition) < 0.5f)
    {
        StartMoving(runDuration);
        EvaluateConditions(isWalking: false);
    }
}

private void StartMoving(float duration)
{
    isMoving = true;
    startTime = Time.time;
    this.duration = duration;
}

private void MovePinocchio()
{
    float timeSinceStarted = Time.time - startTime;
    float fractionOfJourney = timeSinceStarted / duration;
    float smoothFraction = Mathf.SmoothStep(0, 1, fractionOfJourney);
```

```
        transform.position = Vector3.Lerp(startPosition, endPosition, smoothFraction);

        if (fractionOfJourney >= 1.0f)
        {
            isMoving = false;
            Debug.Log("Movement completed.");
        }
    }

    private void EvaluateConditions(bool isWalking)
    {
        Debug.Log("Evaluating light and score conditions.");
        if (lightToggle_204_3.IsGreenLight())
        {
            int countdown = lightToggle_204_3.GetCurrentCountdown();
            if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
            {
                ScoreManager.Instance.AddScore(10); // Gain 10 points
            }
        }
        else
        {
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
```

```
    }
}
```

PinocchioAutoMove_2

```
using UnityEngine;


public class PinocchioAutoMove_2 : MonoBehaviour
{
    public float autoMoveSpeed = 1f;  // Speed for automatic movement, adjustable in the inspector
    public float waitTime = 2f;       // Time to wait at the target position before moving, adjustable in the inspector
    private Vector3 targetPosition = new Vector3(-144, 2, 88);
    private Vector3 checkPosition = new Vector3(-100, 2, 88);
    private bool shouldMove = false;
    private bool isWaiting = false;
    private float startTime;
    private float waitStartTime;
    private float journeyLength;


    void Start()
    {
        journeyLength = Vector3.Distance(checkPosition, targetPosition);
    }


    void Update()
    {
        // Check if Pinocchio is at the specified position and not already moving
        if (Vector3.Distance(transform.position, checkPosition) < 0.01f && !shouldMove && !isWaiting)
        {
            shouldMove = true;
            isWaiting = true;
            waitStartTime = Time.time;  // Start the wait timer
```

```
        }

        // Handle waiting at the check position
        if (isWaiting && (Time.time - waitStartTime >= waitTime))
        {
            isWaiting = false;
            startTime = Time.time;  // Reset start time for movement after waiting
        }

        // Proceed with movement after waiting
        if (shouldMove && !isWaiting)
        {
            float distCovered = (Time.time - startTime) * autoMoveSpeed;
            float fractionOfJourney = distCovered / journeyLength;
            transform.position = Vector3.Lerp(checkPosition, targetPosition,
fractionOfJourney);

            // Check if Pinocchio has reached the target position
            if (Vector3.Distance(transform.position, targetPosition) < 0.01f)
            {
                shouldMove = false;  // Stop moving once the target is reached
            }
        }
    }
}
```

PinocchioMovement_204_2

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204_2 : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(-62, 2, 88);
    private Vector3 endPosition = new Vector3(-100, 2, 88);
    private bool isMoving = false;
    private float startTime;
    private float duration;

    public LightToggle_204 lightToggle_204_2; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition

    void Start()
    {
        Debug.Log("PinocchioMovement_204_2 initialized.");
    }

    void Update()
    {
        if (isMoving)
        {
            MovePinocchio();
        }
    }
```

```
public void StartWalking()
{
    if (Vector3.Distance(transform.position, startPosition) < 0.5f)
    {
        StartMoving(walkDuration);
        EvaluateConditions(isWalking: true);
    }
}

public void StartRunning()
{
    if (Vector3.Distance(transform.position, startPosition) < 0.5f)
    {
        StartMoving(runDuration);
        EvaluateConditions(isWalking: false);
    }
}

private void StartMoving(float duration)
{
    isMoving = true;
    startTime = Time.time;
    this.duration = duration;
}

private void MovePinocchio()
{
    float timeSinceStarted = Time.time - startTime;
    float fractionOfJourney = timeSinceStarted / duration;
    float smoothFraction = Mathf.SmoothStep(0, 1, fractionOfJourney);
```

```
        transform.position = Vector3.Lerp(startPosition, endPosition, smoothFraction);

        if (fractionOfJourney >= 1.0f)
        {
            isMoving = false;
            Debug.Log("Movement completed.");
        }
    }

    private void EvaluateConditions(bool isWalking)
    {
        Debug.Log("Evaluating light and score conditions.");
        if (lightToggle_204_2.IsGreenLight())
        {
            int countdown = lightToggle_204_2.GetCurrentCountdown();
            if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
            {
                ScoreManager.Instance.AddScore(10); // Gain 10 points
            }
        }
        else
        {
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
```

```
    }
}
```

PinocchioAutoMove

```
using UnityEngine;


public class PinocchioAutoMove : MonoBehaviour
{
    public float autoMoveSpeed = 1f;  // Speed for automatic movement, adjustable in the inspector
    public float waitTime = 2f;       // Time to wait at the target position before moving, adjustable in the inspector
    private Vector3 targetPosition = new Vector3(-62, 2, 88);
    private Vector3 checkPosition = new Vector3(-19, 2, 88);
    private bool shouldMove = false;
    private bool isWaiting = false;
    private float startTime;
    private float waitStartTime;
    private float journeyLength;


    void Start()
    {
        journeyLength = Vector3.Distance(checkPosition, targetPosition);
    }


    void Update()
    {
        // Check if Pinocchio is at the specified position and not already moving
        if (Vector3.Distance(transform.position, checkPosition) < 0.01f && !shouldMove && !isWaiting)
        {
            shouldMove = true;
            isWaiting = true;
            waitStartTime = Time.time;  // Start the wait timer
```

```
        }
        // Handle waiting at the check position
        if (isWaiting && (Time.time - waitStartTime >= waitTime))
        {
            isWaiting = false;
            startTime = Time.time;  // Reset start time for movement after waiting
        }
        // Proceed with movement after waiting
        if (shouldMove && !isWaiting)
        {
            float distCovered = (Time.time - startTime) * autoMoveSpeed;
            float fractionOfJourney = distCovered / journeyLength;
            transform.position = Vector3.Lerp(checkPosition, targetPosition,
fractionOfJourney);
            // Check if Pinocchio has reached the target position
            if (Vector3.Distance(transform.position, targetPosition) < 0.01f)
            {
                shouldMove = false;  // Stop moving once the target is reached
            }
        }
    }
}
```

PinocchioMovement_204

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204 : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(19, 2, 88);
    private Vector3 endPosition = new Vector3(-19, 2, 88);
    private bool isMoving = false;
    private float startTime;
    private float speed;

    public LightToggle_204 lightToggle_204; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition
    public AudioClip urgentMoveSound; // Assign a sound clip for urgent movement
under green light with countdown < 5

    void Start()
    {
        transform.position = startPosition;
        Debug.Log("Pinocchio start position set.");
    }

    void Update()
    {
        if (!enabled)
        {
            Debug.Log("PinocchioMovement_204 script is disabled.");
```

```
            return; // Do not execute if the script is disabled
        }

        if (isMoving)
        {
            float timeSinceStarted = Time.time - startTime;
            float fractionOfJourney = timeSinceStarted / speed;
            transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

            Debug.Log("Pinocchio is moving towards end position.");

            // Check if Pinocchio has reached the end position
            if (Vector3.Distance(transform.position, endPosition) < 0.01f)
            {
                isMoving = false;
                Debug.Log("Pinocchio reached the end position.");
            }
        }
    }

    public void StartWalking()
    {
        if (!enabled)
        {
            Debug.Log("Attempt to walk while script is disabled.");
            return; // Do not execute if the script is disabled
        }

        Debug.Log("Pinocchio starts walking.");
        StartMoving(walkDuration);
```

```
    EvaluateConditions(isWalking: true);
}

public void StartRunning()
{
    if (!enabled)
    {
        Debug.Log("Attempt to run while script is disabled.");
        return; // Do not execute if the script is disabled
    }

    Debug.Log("Pinocchio starts running.");
    StartMoving(runDuration);
    EvaluateConditions(isWalking: false);
}

private void StartMoving(float duration)
{
    if (!isMoving)
    {
        isMoving = true;
        startTime = Time.time;
        speed = 1.0f / duration;
        Debug.Log("Movement initialized with duration: " + duration.ToString());
    }
}

private void EvaluateConditions(bool isWalking)
{
    Debug.Log("Evaluating light and score conditions.");
    if (lightToggle_204.IsGreenLight())
```

```
        {
            int countdown = lightToggle_204.GetCurrentCountdown();
            if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
            {
                ScoreManager.Instance.AddScore(10); // Gain 10 points
            }

            // Play urgentMoveSound if the countdown is less than 5
            if (countdown < 5)
            {
                audioSource.PlayOneShot(urgentMoveSound);
                Debug.Log("Playing urgent move sound due to low countdown under green
light.");
            }
        }
        else
        {
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        Debug.Log("Playing sound after delay.");
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 8

IndividualCubeController

```
using System.Collections;
using UnityEngine;
using TMPro;  // Include the TextMeshPro namespace

public class IndividualCubeController : MonoBehaviour
{
   // References to each individual cube, grouped by button
   public GameObject cube5_1, cube5_2, cube5_3, cube5_4, cube5_5;
   public GameObject cube4_1, cube4_2, cube4_3, cube4_4;
   public GameObject cube3_1, cube3_2, cube3_3;
   public GameObject cube2_1, cube2_2;
   public GameObject cube1_1;

   // References to each plane
   public GameObject plane4, plane3, plane2, plane1;
   public GameObject plane2_2, plane2_3, plane2_4, plane2_5; // Additional planes

   // References to TextMeshPro Texts
   public TMP_Text Text_5, Text_4, Text_3, Text_2, Text_1;

   // Public color variable for cube color changes
   public Color newColorForCubes = Color.red;

   void Start()
   {
      DeactivateAll();  // Deactivate all cubes and planes at the start of the scene
      DeactivateTexts();
   }

   public void ShowCubes(int buttonNumber)
   {
      StartCoroutine(ActivateCubesAndPlanes(buttonNumber));
   }

   private IEnumerator ActivateCubesAndPlanes(int buttonNumber)
   {
      DeactivateAll();
      DeactivateTexts();

      if (buttonNumber >= 5)
      {
         Text_5.gameObject.SetActive(true);
         ActivateCubes(cube5_1, cube5_2, cube5_3, cube5_4, cube5_5);
```

```
            yield return new WaitForSeconds(3f);
            plane4.SetActive(true);
        }

        if (buttonNumber >= 4)
        {
            Text_4.gameObject.SetActive(true);
            ActivateCubes(cube4_1, cube4_2, cube4_3, cube4_4);
            yield return new WaitForSeconds(3f);
            plane3.SetActive(true);
        }

        if (buttonNumber >= 3)
        {
            Text_3.gameObject.SetActive(true);
            ActivateCubes(cube3_1, cube3_2, cube3_3);
            yield return new WaitForSeconds(3f);
            plane2.SetActive(true);
        }

        if (buttonNumber >= 2)
        {
            Text_2.gameObject.SetActive(true);
            ActivateCubes(cube2_1, cube2_2);
            yield return new WaitForSeconds(3f);
            plane1.SetActive(true);
        }

        if (buttonNumber >= 1)
        {
            Text_1.gameObject.SetActive(true);
            ActivateCubes(cube1_1);
            yield return new WaitForSeconds(3f);
            StartCoroutine(DelayedPlaneDeactivation());
        }

        yield return new WaitForSeconds(3f); // Additional delay before showing new
planes
        StartCoroutine(ShowAdditionalPlanes(buttonNumber));
    }

    private IEnumerator DelayedPlaneDeactivation()
    {
        yield return new WaitForSeconds(3f);
        plane4.SetActive(false);
        plane3.SetActive(false);
        plane2.SetActive(false);
```

```
    plane1.SetActive(false);
}

private IEnumerator ShowAdditionalPlanes(int buttonNumber)
{
    // Change color of cube1_1 before showing plane2_2
    ChangeCubeColor(cube1_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    plane2_2.SetActive(true);
    yield return new WaitForSeconds(1f);

    // Change colors of subsequent cubes and show planes with delays
    ChangeCubeColor(cube2_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube2_2, newColorForCubes);
    yield return new WaitForSeconds(2f);

    if (buttonNumber >= 3)
    {
        plane2_3.SetActive(true);
        yield return new WaitForSeconds(1f);
    }

    ChangeCubeColor(cube3_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube3_2, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube3_3, newColorForCubes);
    yield return new WaitForSeconds(1f);

    if (buttonNumber >= 4)
    {
        plane2_4.SetActive(true);
        yield return new WaitForSeconds(1f);

        ChangeCubeColor(cube4_1, newColorForCubes);
        yield return new WaitForSeconds(1f);

        ChangeCubeColor(cube4_2, newColorForCubes);
        yield return new WaitForSeconds(1f);

        ChangeCubeColor(cube4_3, newColorForCubes);
        yield return new WaitForSeconds(1f);
```

```
            ChangeCubeColor(cube4_4, newColorForCubes);
        }

        if (buttonNumber == 5)
        {
            plane2_5.SetActive(true);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_1, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_2, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_3, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_4, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_5, newColorForCubes);
        }
    }

    private void DeactivateAll()
    {
        // Deactivate all cubes and planes
        cube5_1.SetActive(false); cube5_2.SetActive(false); cube5_3.SetActive(false);
cube5_4.SetActive(false); cube5_5.SetActive(false);
        cube4_1.SetActive(false); cube4_2.SetActive(false); cube4_3.SetActive(false);
cube4_4.SetActive(false);
        cube3_1.SetActive(false); cube3_2.SetActive(false); cube3_3.SetActive(false);
        cube2_1.SetActive(false); cube2_2.SetActive(false);
        cube1_1.SetActive(false);
        plane4.SetActive(false); plane3.SetActive(false); plane2.SetActive(false);
plane1.SetActive(false);
        plane2_2.SetActive(false); plane2_3.SetActive(false); plane2_4.SetActive(false);
plane2_5.SetActive(false);
    }

    private void DeactivateTexts()
    {
        // Deactivate all texts
        Text_5.gameObject.SetActive(false);
        Text_4.gameObject.SetActive(false);
        Text_3.gameObject.SetActive(false);
```

```
        Text_2.gameObject.SetActive(false);
        Text_1.gameObject.SetActive(false);
    }

    private void ActivateCubes(params GameObject[] cubes)
    {
        foreach (var cube in cubes)
        {
            cube.SetActive(true);
        }
    }

    private void ChangeCubeColor(GameObject cube, Color color)
    {
        Renderer renderer = cube.GetComponent<Renderer>();
        if (renderer != null)
        {
            renderer.material.color = color;
        }
    }
}
```

DESKTOP VRBCS 2

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 1

MoveCubeAround13

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class MoveCubeAround13 : MonoBehaviour
{
    // Directional buttons
    public Button forwardButton;
    public Button rightButton;
    public Button backwardButton;
    public Button leftButton;

    // Magnitude buttons
    public Button magnitudeButton1;
    public Button magnitudeButton2;
    public Button magnitudeButton3;
    public Button magnitudeButton4;
    public Button magnitudeButton5;
    public Button magnitudeButton6;
    public Button magnitudeButton7;
    public Button magnitudeButton8;

    // Run button
    public Button runButton;

    public MoveAround13 targetScript;  // Script that contains the movement logic

    public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

    private bool readyToMoveForward = false;
    private bool readyToMoveRight = false;
    private bool readyToMoveBackward = false;
    private bool readyToMoveLeft = false;
    private int moveMagnitude = 0;  // To store the movement magnitude

    void Start()
    {
        // Assign button click listeners for directions
```

```
    forwardButton.onClick.AddListener(() => SetDirection("forward"));
    rightButton.onClick.AddListener(() => SetDirection("right"));
    backwardButton.onClick.AddListener(() => SetDirection("backward"));
    leftButton.onClick.AddListener(() => SetDirection("left"));

    // Assign button click listeners for magnitudes
    magnitudeButton1.onClick.AddListener(() => SetMagnitude(1));
    magnitudeButton2.onClick.AddListener(() => SetMagnitude(2));
    magnitudeButton3.onClick.AddListener(() => SetMagnitude(3));
    magnitudeButton4.onClick.AddListener(() => SetMagnitude(4));
    magnitudeButton5.onClick.AddListener(() => SetMagnitude(5));
    magnitudeButton6.onClick.AddListener(() => SetMagnitude(6));
    magnitudeButton7.onClick.AddListener(() => SetMagnitude(7));
    magnitudeButton8.onClick.AddListener(() => SetMagnitude(8));

    // Assign button click listener for the run action
    runButton.onClick.AddListener(ProcessMove);
}

private void SetDirection(string direction)
{
    ResetReadyStates();
    switch (direction)
    {
        case "forward":
            readyToMoveForward = true;
            break;
        case "right":
            readyToMoveRight = true;
            break;
        case "backward":
            readyToMoveBackward = true;
            break;
        case "left":
            readyToMoveLeft = true;
            break;
    }
}

private void SetMagnitude(int magnitude)
{
    moveMagnitude = magnitude;
}

private void ProcessMove()
{
    if (moveMagnitude > 0)
```

```
        {
            string message = "";
            if (readyToMoveForward)
            {
                targetScript.MoveCube("forward", moveMagnitude);
                message = $"moveForward({moveMagnitude})";
            }
            else if (readyToMoveRight)
            {
                targetScript.MoveCube("right", moveMagnitude);
                message = $"moveRight({moveMagnitude})";
            }
            else if (readyToMoveBackward)
            {
                targetScript.MoveCube("backward", moveMagnitude);
                message = $"moveBackward({moveMagnitude})";
            }
            else if (readyToMoveLeft)
            {
                targetScript.MoveCube("left", moveMagnitude);
                message = $"moveLeft({moveMagnitude})";
            }

            actionText.text = message;  // Update the UI text
        }
        ResetReadyStates(); // Reset the states after processing the move
    }

    private void ResetReadyStates()
    {
        readyToMoveForward = false;
        readyToMoveRight = false;
        readyToMoveBackward = false;
        readyToMoveLeft = false;
        moveMagnitude = 0;  // Reset magnitude for next input
    }
}
```

MoveAround13

```
using UnityEngine;
using System.Collections;
using TMPro; // Needed for TextMeshPro

public class MoveAround13 : MonoBehaviour
{
   public GameObject cube;               // Assign your Cube GameObject in the
inspector
   public float stepDelay = 3.0f;        // Time to wait at each intermediate step
   public float rotateDuration = 3.0f;    // Time to rotate before moving
   public TextMeshProUGUI counterText;    // Assign your UI Text (TMP) in the
inspector

   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10),
      new Vector3(13, 0.08f, 10),
      new Vector3(12, 0.08f, 10),
      new Vector3(11, 0.08f, 10),
      new Vector3(10, 0.08f, 10),
      new Vector3(9, 0.08f, 10),
      new Vector3(8, 0.08f, 10),
      new Vector3(7, 0.08f, 10),
      new Vector3(6, 0.08f, 10),

      new Vector3(10, 0.08f, 14),
      new Vector3(10, 0.08f, 13),
      new Vector3(10, 0.08f, 12),
      new Vector3(10, 0.08f, 11),

      new Vector3(10, 0.08f, 9),
      new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7),
      new Vector3(10, 0.08f, 6)
   };

   private bool isMoving = false; // Prevent overlapping movements

   // Position helpers (XZ checks with small epsilon)
   private static readonly Vector3 F3 = new Vector3(13f, 0.08f, 10f);
   private static readonly Vector3 F4 = new Vector3(14f, 0.08f, 10f);
   private static readonly Vector3 B3 = new Vector3(7f, 0.08f, 10f);
   private static readonly Vector3 B4 = new Vector3(6f, 0.08f, 10f);
   private const float PosEps = 0.05f;

   void Start()
```

```
    {
        // Start at position 0 with rotation (0, 270, 0)
        cube.transform.position = new Vector3(10f, 0.08f, 10f);
        cube.transform.rotation = Quaternion.Euler(0f, 270f, 0f);

        Debug.Log("Cube Start Position: " + cube.transform.position + " | Start Rotation:
" + cube.transform.rotation.eulerAngles);

        if (counterText != null)
        {
            counterText.text = ""; // Start with empty text
        }
    }

    public void MoveCube(string direction, int magnitude)
    {
        if (isMoving) return; // Ignore if already moving

        Vector3 step = Vector3.zero;

        // Your existing direction mapping:
        switch (direction)
        {
            case "forward": step = Vector3.right; break; // +X  (F1..F4)
            case "right": step = Vector3.back; break; // -Z  (R1..R4)
            case "backward": step = Vector3.left; break; // -X  (B1..B4)
            case "left": step = Vector3.forward; break; // +Z  (L1..L4)
        }

        StartCoroutine(RotateThenMove(step, magnitude));
    }

    // Rotate first (over 3 seconds), then perform your step-by-step movement
    private IEnumerator RotateThenMove(Vector3 step, int magnitude)
    {
        isMoving = true;

        // Determine desired facing based on intended movement direction
        Quaternion targetRotation = DetermineTargetRotation(step);

        // Smoothly rotate from current to target over rotateDuration seconds
        yield return StartCoroutine(RotateTo(targetRotation, rotateDuration));

        // Predict the final position to decide if we need to suppress sounds at F3/B3
and at F4/B4
        Vector3 predictedFinal = PredictFinalPosition(step, magnitude);
```

```
        bool suppressForF4orB4 = ApproximatelyXZ(predictedFinal, F4, PosEps) ||
ApproximatelyXZ(predictedFinal, B4, PosEps);

        // Now proceed with your original stepping logic (with conditional sound
suppression)
        yield return StartCoroutine(MoveStepByStep(step, magnitude,
suppressForF4orB4));

        isMoving = false;
    }

    // Maps movement direction to the requested Y rotations:
    //  Left (+Z / L1..L4)     -> (0,   0, 0)
    //  Right (-Z / R1..R4)    -> (0, 180, 0)
    //  Forward (+X / F1..F4)  -> (0,  90, 0)
    //  Backward (-X / B1..B4) -> (0, 270, 0)
    private Quaternion DetermineTargetRotation(Vector3 step)
    {
        if (step == Vector3.forward) return Quaternion.Euler(0f, 0f, 0f);     // L
        else if (step == Vector3.back) return Quaternion.Euler(0f, 180f, 0f); // R
        else if (step == Vector3.right) return Quaternion.Euler(0f, 90f, 0f); // F
        else if (step == Vector3.left) return Quaternion.Euler(0f, 270f, 0f); // B
        else return cube.transform.rotation;
    }

    // Smooth rotation coroutine
    private IEnumerator RotateTo(Quaternion target, float duration)
    {
        Quaternion startRot = cube.transform.rotation;
        float t = 0f;

        while (t < duration)
        {
            t += Time.deltaTime;
            float lerp = Mathf.Clamp01(t / duration);
            cube.transform.rotation = Quaternion.Slerp(startRot, target, lerp);
            yield return null;
        }

        cube.transform.rotation = target; // snap to final just in case
    }

    // >>> Modified to optionally mute sounds only for runs that end at F4 or B4
    private IEnumerator MoveStepByStep(Vector3 step, int magnitude, bool
suppressForF4orB4)
    {
        for (int i = 0; i < magnitude; i++)
```

```
    {
        Vector3 proposedMove = step;

        if (CanMove(proposedMove))
        {
            cube.transform.position += proposedMove;

            // Update counter
            int countdown = magnitude - (i + 1);
            if (counterText != null)
            {
                counterText.text = countdown.ToString();
            }

            Debug.Log("Moved to " + cube.transform.position + " | Counter: " +
countdown);

            // If the final destination of this command is F4 or B4,
            // then while we're on F3/B3 (passing) or on the final F4/B4,
            // we mute audio during this step's wait so no sound is heard.
            if (suppressForF4orB4)
            {
                bool onF3orB3 = ApproximatelyXZ(cube.transform.position, F3, PosEps) ||
                                ApproximatelyXZ(cube.transform.position, B3, PosEps);
                bool onF4orB4 = ApproximatelyXZ(cube.transform.position, F4, PosEps) ||
                                ApproximatelyXZ(cube.transform.position, B4, PosEps);

                if (onF3orB3 || onF4orB4)
                {
                    float prevVol = AudioListener.volume;
                    AudioListener.volume = 0f;
                    yield return new WaitForSeconds(stepDelay);
                    AudioListener.volume = prevVol;
                    continue;
                }
            }

            // Normal wait
            yield return new WaitForSeconds(stepDelay);
        }
        else
        {
            Debug.Log("Blocked move at " + (cube.transform.position +
proposedMove));
            break; // Stop if blocked
        }
    }
```

```
    }
    // Predict where this command will end given current position, step, and
magnitude
    private Vector3 PredictFinalPosition(Vector3 step, int magnitude)
    {
        Vector3 cursor = cube.transform.position;
        for (int i = 0; i < magnitude; i++)
        {
            Vector3 next = cursor + step;
            if (IsAllowed(next))
                cursor = next;
            else
                break;
        }
        return cursor;
    }
    private bool IsAllowed(Vector3 position)
    {
        for (int i = 0; i < allowedPositions.Length; i++)
        {
            if (Vector3.Distance(position, allowedPositions[i]) < 0.1f)
                return true;
        }
        return false;
    }
    private static bool ApproximatelyXZ(Vector3 a, Vector3 b, float eps)
    {
        return Mathf.Abs(a.x - b.x) <= eps &&
               Mathf.Abs(a.z - b.z) <= eps;
    }

    private bool CanMove(Vector3 proposedMove)
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f)
            {
                return true;
            }
        }

        return false;
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 2 – DESKTOP VRBCS II

CubeStartPosition

```
using UnityEngine;


public class CubeStartPosition : MonoBehaviour
{
    [SerializeField] private Transform cube;  // Assign your cube in the Inspector
    [SerializeField] private Vector3 startPosition = new Vector3(10f, 0.08f, 10f);


    private void Awake()
    {
        if (!cube) cube = transform;  // fallback if nothing assigned
        cube.position = startPosition;
    }
}
```

UserInPutAndScore_2

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using System.Text.RegularExpressions;

public class UserInPutAndScore_2 : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private Transform cube;           // Your cube
    [SerializeField] private TMP_Text scoreText;       // Same TMP Text as the other script

    [Header("UI Buttons (hook these up)")]
    [SerializeField] private Button leftButton;        // LEFT
    [SerializeField] private Button fiveButton;        // 5
    [SerializeField] private Button runButton;         // RUN

    [Header("L2 Detection")]
    [SerializeField, Tooltip("How close the cube must be to count as at L2.")]
    private float atPositionThreshold = 0.05f;

    [Header("Timing")]
    [SerializeField, Tooltip("Seconds to complete Left → 5 → Run after the first button press at L2 (≤0 disables timeout).")]
    private float timeLimitSeconds = 5f;

    [Header("Scoring")]
    [SerializeField, Tooltip("Points this step gives when correct.")]
    private int pointsForThisStep = 10;
```

```
    [SerializeField, Tooltip("Maximum total points across steps (cap).")]
    private int maxTotal = 20;

    [SerializeField, Tooltip("If no existing 'Score: X/Y' is found, assume this prior score.")]
    private int assumedPriorScore = 0;

    [SerializeField, Tooltip("If no existing 'Score: X/Y' is found, assume this prior total.")]
    private int assumedPriorTotal = 10; // so first-time success becomes 10/20

    // --- ADDED ---
    [Header("Audio")]
    [SerializeField] private AudioSource audioSource;   // Drag an AudioSource here
    [SerializeField] private AudioClip successClip;     // Drag the 20/20 sound here
    // --- ADDED ---

    private static readonly Vector3 L2 = new Vector3(10f, 0.08f, 12f);

    private enum Step { ExpectLeft, ExpectFive, ExpectRun, Done }
    private Step step = Step.ExpectLeft;

    private bool success = false;
    private bool failed = false;

    // countdown state
    private bool countdownActive = false;
    private float timeRemaining = 0f;

    private void Awake()
    {
```

```
    if (!cube) cube = transform;

    if (leftButton) leftButton.onClick.AddListener(OnLeftClicked);
    if (fiveButton) fiveButton.onClick.AddListener(OnFiveClicked);
    if (runButton) runButton.onClick.AddListener(OnRunClicked);
}

private void OnDestroy()
{
    if (leftButton) leftButton.onClick.RemoveListener(OnLeftClicked);
    if (fiveButton) fiveButton.onClick.RemoveListener(OnFiveClicked);
    if (runButton) runButton.onClick.RemoveListener(OnRunClicked);
}

private void Update()
{
    if (!countdownActive || success || failed || step == Step.Done) return;

    timeRemaining -= Time.deltaTime;
    if (timeRemaining <= 0f)
    {
        Fail(); // time's up → +0 score, total increases only up to maxTotal
    }
}

private bool AtL2()
{
    return Vector3.Distance(cube.position, L2) <= atPositionThreshold;
}

public void OnLeftClicked() => HandlePress(ButtonKind.Left);
```

```
public void OnFiveClicked() => HandlePress(ButtonKind.Five);
public void OnRunClicked() => HandlePress(ButtonKind.Run);

private enum ButtonKind { Left, Five, Run }

private void HandlePress(ButtonKind kind)
{
    if (!AtL2() || success || failed || step == Step.Done) return;

    // Start the countdown on the first eligible press (if enabled)
    if (!countdownActive && timeLimitSeconds > 0f)
    {
        countdownActive = true;
        timeRemaining = timeLimitSeconds;
    }

    switch (step)
    {
        case Step.ExpectLeft:
            if (kind == ButtonKind.Left) step = Step.ExpectFive;
            else Fail();
            break;

        case Step.ExpectFive:
            if (kind == ButtonKind.Five) step = Step.ExpectRun;
            else Fail();
            break;

        case Step.ExpectRun:
            if (kind == ButtonKind.Run) Succeed();
            else Fail();
```

```
            break;
        }
    }

    private void Succeed()
    {
        success = true;
        step = Step.Done;
        countdownActive = false;
        // +10 to score, attempt +10 to total but cap total at maxTotal
        AddToScore(pointsForThisStep, pointsForThisStep);
    }

    private void Fail()
    {
        failed = true;
        step = Step.Done;
        countdownActive = false;
        // +0 to score, attempt +10 to total but cap total at maxTotal
        AddToScore(0, pointsForThisStep);
    }

    private void AddToScore(int deltaScore, int deltaTotal)
    {
        if (!scoreText)
        {
            Debug.Log($"Would update score by +{deltaScore}/+{deltaTotal}, but no
ScoreText is assigned.");
            return;
        }
```

```
    // Parse current "X/Y"; if none, use assumed prior (0/10)
    int currentScore, currentTotal;
    if (!TryParseScore(scoreText.text, out currentScore, out currentTotal))
    {
        currentScore = Mathf.Max(0, assumedPriorScore);
        currentTotal = Mathf.Max(0, assumedPriorTotal);
    }

    // Only increase total up to the cap (maxTotal)
    int allowedTotalIncrease = Mathf.Max(0, maxTotal - currentTotal);
    int totalIncrease = Mathf.Clamp(deltaTotal, 0, allowedTotalIncrease);

    int newTotal = currentTotal + totalIncrease;

    // Increase score, but never beyond the (possibly capped) total
    int newScore = Mathf.Clamp(currentScore + deltaScore, 0, newTotal);

    scoreText.gameObject.SetActive(true);
    scoreText.text = $"Score: {newScore}/{newTotal}";

    // --- ADDED: play sound exactly when it becomes 20/20 ---
    if (newScore == 20 && newTotal == 20 && audioSource)
    {
        if (successClip) audioSource.PlayOneShot(successClip);
        else audioSource.Play();
    }
    // --- END ADDED ---
}

private static bool TryParseScore(string input, out int score, out int total)
{
```

```
        score = 0;
        total = 0;
        if (string.IsNullOrEmpty(input)) return false;

        var m = Regex.Match(input, @"(\d+)\s*/\s*(\d+)");
        if (!m.Success) return false;

        int.TryParse(m.Groups[1].Value, out score);
        int.TryParse(m.Groups[2].Value, out total);
        return true;
    }

    public void ResetSequence(bool hideScore = false)
    {
        step = Step.ExpectLeft;
        success = false;
        failed = false;
        countdownActive = false;
        timeRemaining = 0f;

        if (scoreText && hideScore) scoreText.gameObject.SetActive(false);
    }
}
```

InstructionsTexts_2

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

public class InstructionsTexts_2 : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private Transform cube;                // Drag your cube here
    [SerializeField] private TMP_Text instructionText3;     // TMP Text for Instruction 3
    [SerializeField] private TMP_Text instructionText4;     // TMP Text for Instruction 4

    [Header("Timing")]
    [SerializeField, Tooltip("How long after reaching R3 before showing InstructionText_3")]
    private float delayAtR3 = 1.5f;                         // <<< X seconds in Inspector
    [SerializeField, Tooltip("How long InstructionText_3 stays after reaching L2 before showing InstructionText_4")]
    private float extraTimeAtL2 = 2f;

    [Header("Detection")]
    [SerializeField, Tooltip("How close the cube must be to count as at R3 or L2")]
    private float atPositionThreshold = 0.05f;

    // Exact positions (USE THESE)
    private static readonly Vector3 R3 = new Vector3(10f, 0.08f, 7f);
    private static readonly Vector3 L2 = new Vector3(10f, 0.08f, 12f);

    private enum Phase { Idle, WaitingR3Delay, Showing3, WaitingExtra, Showing4 }
    private Phase currentPhase = Phase.Idle;
```

```
private bool wasAtR3, wasAtL2;
private Coroutine r3DelayCoroutine;
private Coroutine postL2Coroutine;

private void Awake()
{
    if (!cube) cube = transform;

    // Hide both texts initially
    if (instructionText3) instructionText3.gameObject.SetActive(false);
    if (instructionText4) instructionText4.gameObject.SetActive(false);
}

private void Update()
{
    bool atR3 = Vector3.Distance(cube.position, R3) <= atPositionThreshold;
    bool atL2 = Vector3.Distance(cube.position, L2) <= atPositionThreshold;

    // Arrived at R3 -> start delay countdown for text3
    if (atR3 && !wasAtR3)
    {
        StartR3Delay();
    }

    // Left R3 while waiting -> cancel delay
    if (!atR3 && wasAtR3 && currentPhase == Phase.WaitingR3Delay)
    {
        CancelR3Delay();
    }
```

```
    // Arrived at L2 after text3 is visible -> start countdown for text4
    if (atL2 && !wasAtL2 && currentPhase == Phase.Showing3)
    {
        StartExtraTimeAtL2();
    }

    wasAtR3 = atR3;
    wasAtL2 = atL2;
}

private void StartR3Delay()
{
    currentPhase = Phase.WaitingR3Delay;

    // Hide texts while waiting
    if (instructionText3) instructionText3.gameObject.SetActive(false);
    if (instructionText4) instructionText4.gameObject.SetActive(false);

    if (r3DelayCoroutine != null) StopCoroutine(r3DelayCoroutine);
    r3DelayCoroutine = StartCoroutine(ShowText3AfterStayingAtR3(delayAtR3));
}

private void CancelR3Delay()
{
    if (r3DelayCoroutine != null)
    {
        StopCoroutine(r3DelayCoroutine);
        r3DelayCoroutine = null;
    }
    currentPhase = Phase.Idle;
}
```

```
private IEnumerator ShowText3AfterStayingAtR3(float seconds)
{
    float t = 0f;

    // Require continuous presence at R3 for the full duration
    while (t < seconds)
    {
        if (Vector3.Distance(cube.position, R3) > atPositionThreshold)
        {
            currentPhase = Phase.Idle;
            r3DelayCoroutine = null;
            yield break; // Abort if we drift away
        }
        t += Time.deltaTime;
        yield return null;
    }

    r3DelayCoroutine = null;
    StartShowingText3();
}

private void StartShowingText3()
{
    if (postL2Coroutine != null) StopCoroutine(postL2Coroutine);

    // Show text3, hide text4
    if (instructionText3) instructionText3.gameObject.SetActive(true);
    if (instructionText4) instructionText4.gameObject.SetActive(false);

    currentPhase = Phase.Showing3;
```

```
    }

    private void StartExtraTimeAtL2()
    {
        currentPhase = Phase.WaitingExtra;
        if (postL2Coroutine != null) StopCoroutine(postL2Coroutine);
        postL2Coroutine = StartCoroutine(SwitchToText4AfterDelay());
    }

    private IEnumerator SwitchToText4AfterDelay()
    {
        yield return new WaitForSeconds(extraTimeAtL2);

        // Hide text3, show text4
        if (instructionText3) instructionText3.gameObject.SetActive(false);
        if (instructionText4) instructionText4.gameObject.SetActive(true);

        currentPhase = Phase.Showing4;
    }
}
```

InstructionsTexts

```
using UnityEngine;
using TMPro;
using System.Collections;
using UnityEngine.UI; // ✨ NEW

public class InstructionsTexts : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private Transform cube;                // Drag your cube here
    [SerializeField] private TMP_Text instructionText1;     // TMP Text for Instruction 1
    [SerializeField] private TMP_Text instructionText2;     // TMP Text for Instruction 2
    [SerializeField] private Button buttonStart2;           // ✨ NEW: assign
Button_START_2 here (optional, see Awake)

    [Header("Timing")]
    [SerializeField, Tooltip("Time after reaching r3 before showing InstructionText_2")]
    private float extraTimeAtR3 = 2f;

    [SerializeField, Tooltip("Time after InstructionText_2 appears before hiding it")]
    private float hideText2Delay = 5f;   // Set in the Unity Inspector

    [Header("Detection")]
    [SerializeField, Tooltip("How close the cube must be to count as at 0 or r3")]
    private float atPositionThreshold = 0.05f;

    // Exact positions
    private static readonly Vector3 P0 = new Vector3(10f, 0.08f, 10f);
    private static readonly Vector3 R3 = new Vector3(10f, 0.08f, 7f);

    private enum Phase { Idle, Showing1, WaitingExtra, Showing2 }
```

```
private Phase currentPhase = Phase.Idle;

private bool wasAtP0, wasAtR3;
private bool text1Locked;
private Coroutine postR3Coroutine;
private Coroutine hideText2Coroutine;

// ✨ NEW: master kill-switch once Start_2 is pressed
private bool instructionsDisabled = false;

private void Awake()
{
    if (!cube) cube = transform;

    if (instructionText1) instructionText1.gameObject.SetActive(false);
    if (instructionText2) instructionText2.gameObject.SetActive(false);

    // ✨ NEW: If you assign the button in the Inspector, wire it automatically.
    if (buttonStart2 != null)
    {
        buttonStart2.onClick.AddListener(OnStart2Pressed);
    }
}

private void Update()
{
    // ✨ NEW: if disabled, never show instructions again
    if (instructionsDisabled) return;

    bool atP0 = Vector3.Distance(cube.position, P0) <= atPositionThreshold;
```

```
    bool atR3 = Vector3.Distance(cube.position, R3) <= atPositionThreshold;

    // Arrive at P0 -> show InstructionText_1 (only if it hasn't been locked)
    if (atP0 && !wasAtP0 && !text1Locked)
    {
        StartShowingText1();
    }

    // Arrive at R3
    if (atR3 && !wasAtR3 && currentPhase == Phase.Showing1)
    {
        StartExtraTimeAtR3(); // After delay, show text2 and start hide timer
    }

    wasAtP0 = atP0;
    wasAtR3 = atR3;
}

private void StartShowingText1()
{
    if (postR3Coroutine != null) StopCoroutine(postR3Coroutine);

    if (instructionText1) instructionText1.gameObject.SetActive(true);
    if (instructionText2) instructionText2.gameObject.SetActive(false);

    currentPhase = Phase.Showing1;
}

private void StartExtraTimeAtR3()
{
    currentPhase = Phase.WaitingExtra;
```

```
    if (postR3Coroutine != null) StopCoroutine(postR3Coroutine);
    postR3Coroutine = StartCoroutine(SwitchToText2AfterDelay());
}

private IEnumerator SwitchToText2AfterDelay()
{
    yield return new WaitForSeconds(extraTimeAtR3);

    // Hide text1 permanently and show text2
    if (instructionText1) instructionText1.gameObject.SetActive(false);
    text1Locked = true;

    if (instructionText2) instructionText2.gameObject.SetActive(true);

    currentPhase = Phase.Showing2;

    // Start the hide timer NOW (after text2 is visible)
    if (hideText2Coroutine != null) StopCoroutine(hideText2Coroutine);
    hideText2Coroutine = StartCoroutine(HideText2AfterDelay());
}

private IEnumerator HideText2AfterDelay()
{
    float elapsed = 0f;
    while (elapsed < hideText2Delay)
    {
        // If cube leaves R3 before timer finishes, stop
        if (Vector3.Distance(cube.position, R3) > atPositionThreshold)
            yield break;

        elapsed += Time.deltaTime;
```

```
            yield return null;
        }

        if (instructionText2) instructionText2.gameObject.SetActive(false);
        currentPhase = Phase.Idle;
        hideText2Coroutine = null;
    }

    // ✨ NEW: Call this from Button_START_2 (via OnClick) to permanently suppress the instructions.
    public void OnStart2Pressed()
    {
        instructionsDisabled = true;

        // Stop any running coroutines
        if (postR3Coroutine != null) { StopCoroutine(postR3Coroutine); postR3Coroutine = null; }
        if (hideText2Coroutine != null) { StopCoroutine(hideText2Coroutine); hideText2Coroutine = null; }

        // Hide any visible texts
        if (instructionText1) instructionText1.gameObject.SetActive(false);
        if (instructionText2) instructionText2.gameObject.SetActive(false);

        // Lock state so nothing re-triggers even if you later re-enable Update logic
        text1Locked = true;
        currentPhase = Phase.Idle;
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 3 – DESKTOP VRBCS II

CubeController

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

public class CubeController : MonoBehaviour
{
   [Header("Grid")]
   [SerializeField] private int gridSize = 8;

   [Header("Movement")]
   [Min(0f)] [SerializeField] private float stepDuration = 0.15f;
   [Min(0f)] [SerializeField] private float dwellTime = 0.20f;
   [SerializeField] private bool smooth = true;

   [Header("UI Counters")]
   public TextMeshProUGUI counterXText;
   public TextMeshProUGUI counterYText;

   private Coroutine pathRoutine;

   void Start()
   {
      transform.position = new Vector3(0f, 0f, 0f);
      UpdateCounters(0, 0);
   }

   public void TraverseTo(int targetX, int targetZ)
   {
      targetX = Mathf.Clamp(targetX, 0, gridSize - 1);
      targetZ = Mathf.Clamp(targetZ, 0, gridSize - 1);

      if (pathRoutine != null) StopCoroutine(pathRoutine);
      pathRoutine = StartCoroutine(TraversePath(targetX, targetZ));
   }

   private IEnumerator TraversePath(int targetX, int targetZ)
   {
      yield return MoveToCell(new Vector3(0f, 0f, 0f), true);

      for (int x = 0; x < targetX; x++)
      {
         for (int z = 0; z < gridSize; z++)
            yield return MoveToCell(new Vector3(x, 0f, z));
```

```
        }

        for (int z = 0; z <= targetZ; z++)
            yield return MoveToCell(new Vector3(targetX, 0f, z));

        pathRoutine = null;
    }

    private IEnumerator MoveToCell(Vector3 destination, bool instantIfAlreadyThere =
false)
    {
        int cx = (int)destination.x;
        int cy = (int)destination.z;

        if (instantIfAlreadyThere && transform.position == destination)
        {
            UpdateCounters(cx, cy);
            yield break;
        }

        if (!smooth || stepDuration <= 0f)
        {
            transform.position = destination;
        }
        else
        {
            Vector3 start = transform.position;
            float t = 0f;
            while (t < 1f)
            {
                t += Time.deltaTime / stepDuration;
                transform.position = Vector3.Lerp(start, destination, Mathf.Clamp01(t));
                yield return null;
            }
            transform.position = destination;
        }

        UpdateCounters(cx, cy);

        if (dwellTime > 0f)
            yield return new WaitForSeconds(dwellTime);
        else
            yield return null;
    }

    private void UpdateCounters(int cx, int cy)
    {
```

```
        if (counterXText != null) counterXText.text = "Counter-X  i  " + cx;
        if (counterYText != null) counterYText.text = "Counter-Y  j  " + cy;
    }
}
```

ButtonManager

```
using UnityEngine;
using TMPro;

public class ButtonManager : MonoBehaviour
{
   public CubeController cubeController;
   public TextMeshProUGUI moveText;

   private int? xSelection = null;
   private int? zSelection = null;

   public void OnButtonPress(int buttonNumber)
   {
      if (buttonNumber < 8) xSelection = buttonNumber;
      else if (buttonNumber < 16) zSelection = buttonNumber - 8;

      if (xSelection.HasValue && zSelection.HasValue)
      {
         int x = xSelection.Value;
         int z = zSelection.Value;
         moveText.text = $"Move({x}, {z})";
         cubeController.TraverseTo(x, z);
         xSelection = null;
         zSelection = null;
      }
   }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 4 – DESKTOP VRBCS II

AutoMoveStart1

```
using UnityEngine;

using UnityEngine.UI;

using TMPro;

using System.Collections;


public class AutoMoveStart1 : MonoBehaviour

{

   [Header("Refs")]

   [SerializeField] private CubeController16 cubeController;


   [Header("UI: Instructions")]

   [SerializeField] private GameObject instructionsText3; // show 10s after cube at (3,0,4)

   [SerializeField] private GameObject instructionsText4; // show when cube at (1,0,3) but ONLY after (3,0,4) was reached


   [Header("UI: Score (TMP)")]

   [SerializeField] private TMP_Text scoreText;


   [Header("UI: Buttons for this step (Order X1 -> Y3)")]

   [SerializeField] private Button buttonX1;

   [SerializeField] private Button buttonY3;


   [Header("Targets & Timings")]

   [SerializeField] private Vector3 targetPosStep1 = new Vector3(1f, 0f, 3f);     // for Text_4

   [SerializeField] private Vector3 triggerPosForText3 = new Vector3(3f, 0f, 4f); // for Text_3

   [SerializeField] private float posTolerance = 0.01f;
```

```
    [SerializeField] private float text3DelaySeconds = 10f;

    private bool start1Pressed = false;
    private bool atTargetStep1 = false;
    private bool listeningForOrder = false;

    private bool pressedX1 = false; // must come first
    private bool pressedY3 = false;

    private bool attemptCompleted = false;

    // tracking for Instructions_Text_3 trigger at (3,0,4)
    private bool reachedTriggerForText3 = false; // becomes true once we detect cube
at (3,0,4)
    private bool text3Shown = false;
    private Coroutine text3DelayRoutine;

    private void Start()
    {
        // Ensure instruction texts are hidden at start
        if (instructionsText3 != null) instructionsText3.SetActive(false);
        if (instructionsText4 != null) instructionsText4.SetActive(false);
    }

    private void OnEnable()
    {
        if (buttonX1 != null) buttonX1.onClick.AddListener(OnButtonX1Pressed);
        if (buttonY3 != null) buttonY3.onClick.AddListener(OnButtonY3Pressed);
    }

    private void OnDisable()
```

```
    {
        if (buttonX1 != null) buttonX1.onClick.RemoveListener(OnButtonX1Pressed);
        if (buttonY3 != null) buttonY3.onClick.RemoveListener(OnButtonY3Pressed);
    }

    private void Update()
    {
        if (cubeController == null) return;

        // --- Watch for (3,0,4) to start delayed show of Instructions_Text_3 ---
        if (!text3Shown && !reachedTriggerForText3 &&
            Vector3.Distance(cubeController.transform.position, triggerPosForText3) <=
posTolerance)
        {
            reachedTriggerForText3 = true;
            if (text3DelayRoutine != null) StopCoroutine(text3DelayRoutine);
            text3DelayRoutine = StartCoroutine(ShowText3AfterDelay());
        }

        // --- After Start_1, watch for (1,0,3) to switch Text_3 -> Text_4 ---
        // IMPORTANT: Only allow Text_4 if the cube has ALREADY reached (3,0,4) at
least once.
        if (start1Pressed && !atTargetStep1 && reachedTriggerForText3 &&
            Vector3.Distance(cubeController.transform.position, targetPosStep1) <=
posTolerance)
        {
            atTargetStep1 = true;

            // Hide Text_3 and cancel any pending delayed show
            if (text3DelayRoutine != null)
            {
                StopCoroutine(text3DelayRoutine);
```

```
            text3DelayRoutine = null;
        }
        text3Shown = true; // lock it so it won't be scheduled again
        if (instructionsText3 != null) instructionsText3.SetActive(false);

        // Show Text_4 now
        if (instructionsText4 != null) instructionsText4.SetActive(true);

        // Begin listening for X1 -> Y3
        listeningForOrder = true;
        pressedX1 = false;
        pressedY3 = false;
        attemptCompleted = false;
    }
}

private IEnumerator ShowText3AfterDelay()
{
    yield return new WaitForSeconds(text3DelaySeconds);
    if (!text3Shown) // still not shown and delay elapsed
    {
        text3Shown = true;
        if (instructionsText3 != null) instructionsText3.SetActive(true);
    }
}

// Hook this to the Start_1 UI Button
public void OnStart1ButtonPressed()
{
    if (cubeController == null)
    {
```

```
            Debug.LogWarning("CubeController16 reference is missing!");
            return;
        }

        // Reset step-1 state
        start1Pressed = true;
        atTargetStep1 = false;
        listeningForOrder = false;
        attemptCompleted = false;
        pressedX1 = false;
        pressedY3 = false;

        // We do NOT reset the Text_3 (3,0,4) tracker here, since it's global.
        // If you want to fully reset the (3,0,4) / 10s logic on each start, uncomment the next lines:
        // reachedTriggerForText3 = false;
        // text3Shown = false;
        // if (text3DelayRoutine != null) { StopCoroutine(text3DelayRoutine); text3DelayRoutine = null; }
        // if (instructionsText3 != null) instructionsText3.SetActive(false);

        // Prepare instructions visibility for this step
        if (instructionsText4 != null) instructionsText4.SetActive(false);

        // Reset cube and move to (1,0,3)
        cubeController.transform.position = Vector3.zero;
        cubeController.TraverseTo(1, 3);
    }

    private void OnButtonX1Pressed()
    {
```

```
    if (!listeningForOrder || attemptCompleted) return;

    if (!pressedX1 && !pressedY3)
    {
        pressedX1 = true;
        TryCompleteIfSuccess();
    }
}

private void OnButtonY3Pressed()
{
    if (!listeningForOrder || attemptCompleted) return;

    if (pressedX1 && !pressedY3)
    {
        pressedY3 = true;
        TryCompleteIfSuccess();
    }
}

private void TryCompleteIfSuccess()
{
    if (pressedX1 && pressedY3)
    {
        attemptCompleted = true;
        listeningForOrder = false;

        // Add +10 to existing score (capped at 20)
        int current, max;
        (current, max) = ParseScore(scoreText != null ? scoreText.text : null, 20);
        current = Mathf.Min(current + 10, max);
```

```
            if (scoreText != null) scoreText.text = $"Score: {current}/{max}";
        }
    }

    private (int current, int max) ParseScore(string input, int defaultMax)
    {
        if (string.IsNullOrEmpty(input))
            return (0, defaultMax);

        var m = System.Text.RegularExpressions.Regex.Matches(input, @"\d+");
        int found = 0, first = 0, second = defaultMax;
        foreach (System.Text.RegularExpressions.Match mm in m)
        {
            if (!mm.Success) continue;
            if (found == 0) { int.TryParse(mm.Value, out first); found = 1; }
            else { int.TryParse(mm.Value, out second); break; }
        }
        if (found == 0) return (0, defaultMax);
        if (found == 1) return (first, defaultMax);
        return (first, Mathf.Max(1, second));
    }
}
```

AutoMoveStart

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

public class AutoMoveStart : MonoBehaviour
{
    [SerializeField] private CubeController16 cubeController;
    [SerializeField] private GameObject instructionText1;
    [SerializeField] private GameObject instructionText2;
    [SerializeField] private TMP_Text scoreText;   // TMP text for the score
    [SerializeField] private Button buttonX3;      // Assign in Inspector
    [SerializeField] private Button buttonY4;      // Assign in Inspector

    private bool cubeAtTarget = false;
    private bool pressedX3 = false;
    private bool pressedY4 = false;
    private Coroutine checkCoroutine;

    private void Start()
    {
        // Show Instruction 1 at the start
        if (instructionText1 != null) instructionText1.SetActive(true);
        if (instructionText2 != null) instructionText2.SetActive(false);

        // Register button listeners
        if (buttonX3 != null) buttonX3.onClick.AddListener(OnButtonX3Pressed);
        if (buttonY4 != null) buttonY4.onClick.AddListener(OnButtonY4Pressed);
    }
```

```
    private void Update()
    {
        if (cubeController != null && !cubeAtTarget)
        {
            if (Vector3.Distance(cubeController.transform.position, new Vector3(3f, 0f, 4f))
< 0.01f)
            {
                cubeAtTarget = true;

                if (instructionText1 != null) instructionText1.SetActive(false);
                if (instructionText2 != null) instructionText2.SetActive(true);

                // Start the 10s check
                checkCoroutine = StartCoroutine(CheckButtonOrderAfterDelay());
            }
        }
    }

    private IEnumerator CheckButtonOrderAfterDelay()
    {
        yield return new WaitForSeconds(10f);

        // If player hasn't succeeded yet → score = 0
        if (!(pressedX3 && pressedY4))
        {
            if (scoreText != null) scoreText.text = "Score: 0/20";
        }
    }

    private void OnButtonX3Pressed()
```

```
{
    HideInstruction2(); // 👈 Hide instructions immediately

    if (!pressedX3 && !pressedY4) pressedX3 = true;
    CheckIfSuccess();
}

private void OnButtonY4Pressed()
{
    HideInstruction2(); // 👈 Hide instructions immediately

    if (pressedX3 && !pressedY4) pressedY4 = true;
    CheckIfSuccess();
}

private void CheckIfSuccess()
{
    if (pressedX3 && pressedY4)
    {
        if (scoreText != null) scoreText.text = "Score: 10/20";

        // Stop the coroutine so it doesn't overwrite with 0 later
        if (checkCoroutine != null) StopCoroutine(checkCoroutine);
    }
}

private void HideInstruction2()
{
    if (instructionText2 != null && instructionText2.activeSelf)
    {
        instructionText2.SetActive(false);
```

```
        }
    }

    // This function will be called when the Start button is pressed
    public void OnStartButtonPressed()
    {
        if (cubeController != null)
        {
            cubeController.TraverseTo(3, 4);
        }
        else
        {
            Debug.LogWarning("CubeController16 reference is missing!");
        }
    }
}
```

CubeController16

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

public class CubeController16 : MonoBehaviour
{
    [Header("Grid")]
    [SerializeField] private int gridSize = 8;

    [Header("Movement")]
    [Min(0f)] [SerializeField] private float stepDuration = 0.15f;
    [Min(0f)] [SerializeField] private float dwellTime = 0.20f;
    [SerializeField] private bool smooth = true;

    [Header("UI Counters")]
    public TextMeshProUGUI counterXText;
    public TextMeshProUGUI counterYText;

    private Coroutine pathRoutine;

    void Start()
    {
        transform.position = new Vector3(0f, 0f, 0f);
        UpdateCounters(0, 0);
    }

    public void TraverseTo(int targetX, int targetZ)
    {
        targetX = Mathf.Clamp(targetX, 0, gridSize - 1);
```

```
        targetZ = Mathf.Clamp(targetZ, 0, gridSize - 1);

        if (pathRoutine != null) StopCoroutine(pathRoutine);
        pathRoutine = StartCoroutine(TraversePath(targetX, targetZ));
    }

    private IEnumerator TraversePath(int targetX, int targetZ)
    {
        yield return MoveToCell(new Vector3(0f, 0f, 0f), true);

        for (int x = 0; x < targetX; x++)
        {
            for (int z = 0; z < gridSize; z++)
                yield return MoveToCell(new Vector3(x, 0f, z));
        }

        for (int z = 0; z <= targetZ; z++)
            yield return MoveToCell(new Vector3(targetX, 0f, z));

        pathRoutine = null;
    }

    private IEnumerator MoveToCell(Vector3 destination, bool instantIfAlreadyThere =
false)
    {
        int cx = (int)destination.x;
        int cy = (int)destination.z;

        if (instantIfAlreadyThere && transform.position == destination)
        {
            UpdateCounters(cx, cy);
```

```
            yield break;
        }

        if (!smooth || stepDuration <= 0f)
        {
            transform.position = destination;
        }
        else
        {
            Vector3 start = transform.position;
            float t = 0f;
            while (t < 1f)
            {
                t += Time.deltaTime / stepDuration;
                transform.position = Vector3.Lerp(start, destination, Mathf.Clamp01(t));
                yield return null;
            }
            transform.position = destination;
        }

        UpdateCounters(cx, cy);

        if (dwellTime > 0f)
            yield return new WaitForSeconds(dwellTime);
        else
            yield return null;
    }

    private void UpdateCounters(int cx, int cy)
    {
        if (counterXText != null) counterXText.text = "Counter-X  i  " + cx;
```

```
        if (counterYText != null) counterYText.text = "Counter-Y  j  " + cy;
    }
}
```

ButtonManager16

```
using UnityEngine;
using TMPro;

public class ButtonManager16 : MonoBehaviour
{
    public CubeController16 cubeController;
    public TextMeshProUGUI moveText;

    private int? xSelection = null;
    private int? zSelection = null;

    public void OnButtonPress(int buttonNumber)
    {
        if (buttonNumber < 8) xSelection = buttonNumber;
        else if (buttonNumber < 16) zSelection = buttonNumber - 8;
        if (xSelection.HasValue && zSelection.HasValue)
        {
            int x = xSelection.Value;
            int z = zSelection.Value;
            moveText.text = $"Move({x}, {z})";
            cubeController.TraverseTo(x, z);
            xSelection = null;
            zSelection = null;
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 5 – DESKTOP VRBCS II

PinocchioMovement17

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

public class PinocchioMovement17 : MonoBehaviour
{
    [Header("Movement")]
    public Vector3 startPosition = new Vector3(18.5f, 2f, 88f);
    public Vector3 endPosition = new Vector3(-19f, 2f, 88f);

    [Tooltip("Reference to the LightToggle17 in the scene.")]
    public LightToggle17 lightToggle;

    [Header("Arrival Behavior")]
    [Tooltip("How long Pinocchio stays at the end before disappearing.")]
    public float endHoldSeconds = 2f;

    [Header("Movement Durations")]
    [Tooltip("Seconds to travel from start to end for the FIRST run.")]
    public float firstMoveDuration = 10f;

    [Tooltip("Seconds to travel from start to end for the SECOND and later runs.")]
    public float secondMoveDuration = 5f;

    // Internal state
    private float t;
    private bool moving;
```

```
private bool firstRunStarted;
private bool waitingForCountdown7;
private int lastSeenCountdown = int.MinValue;

private float currentMoveDuration;

// NEW: run tracking
private int completedRuns = 0;
private bool finishedAllRuns = false;

// Renderers to toggle visibility without disabling the GameObject
private List<Renderer> renderers = new List<Renderer>();

private void Awake()
{
   transform.position = startPosition;
   CacheRenderers();
}

private void OnEnable()
{
   if (lightToggle != null)
   {
      lightToggle.OnGreenStarted += HandleGreenStarted;
      lightToggle.OnRedStarted += HandleRedStarted;
   }
}

private void OnDisable()
{
   if (lightToggle != null)
```

```
    {
        lightToggle.OnGreenStarted -= HandleGreenStarted;
        lightToggle.OnRedStarted -= HandleRedStarted;
    }
}

private void Update()
{
    if (lightToggle == null || finishedAllRuns) return;

    // First run: wait for GREEN and countdown == 13
    if (!firstRunStarted)
    {
        int c = lightToggle.GetCurrentCountdown();
        if (lightToggle.IsGreenLight() && c != lastSeenCountdown && c == 13)
        {
            firstRunStarted = true;
            currentMoveDuration = Mathf.Max(0.0001f, firstMoveDuration);
            BeginMoveFromStart();
            completedRuns = 1; // mark we're doing the first run
        }
        lastSeenCountdown = c;
    }
    // After reset: wait for GREEN and countdown == 7 (second run)
    else if (waitingForCountdown7)
    {
        int c = lightToggle.GetCurrentCountdown();
        if (lightToggle.IsGreenLight() && c != lastSeenCountdown && c == 7)
        {
            currentMoveDuration = Mathf.Max(0.0001f, secondMoveDuration);
            BeginMoveFromStart();
```

```
            completedRuns = 2; // mark we're doing the second run
        }
        lastSeenCountdown = c;
    }

    // Advance continuously once started
    if (moving)
    {
        t += Time.deltaTime / currentMoveDuration;
        t = Mathf.Clamp01(t);
        transform.position = Vector3.Lerp(startPosition, endPosition, t);

        if (t >= 1f - 1e-5f)
        {
            StartCoroutine(HandleArrivalAtEnd());
        }
    }
}

private void HandleGreenStarted()
{
    // Starts are handled via countdown checks in Update.
}

private void HandleRedStarted()
{
    // Movement continues once started.
}

private void BeginMoveFromStart()
{
```

```
        t = 0f;
        transform.position = startPosition;
        Show(true);

        moving = true;
        waitingForCountdown7 = false;
    }

    private IEnumerator HandleArrivalAtEnd()
    {
        moving = false;
        transform.position = endPosition;

        // If we've completed the second movement, stop here permanently.
        if (completedRuns >= 2)
        {
            finishedAllRuns = true;
            yield break; // do NOT reset; stay at (-19, 2, 88)
        }

        // Otherwise (after first run), do your existing reset flow and prepare for second
run
        yield return new WaitForSeconds(endHoldSeconds);

        Show(false);
        transform.position = startPosition;
        yield return null;
        Show(true);

        // Next run starts at GREEN + countdown == 7
        waitingForCountdown7 = true;
```

```
    }

    private void CacheRenderers()
    {
        renderers.Clear();
        GetComponentsInChildren(true, renderers);
    }

    private void Show(bool visible)
    {
        foreach (var r in renderers)
        {
            if (r != null) r.enabled = visible;
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 6 – DESKTOP VRBCS II

LightToggle

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

public class LightToggle : MonoBehaviour
{
    public Light greenLight;
    public Light redLight;
    public TextMeshProUGUI countdownText;
    public float delay = 10f;  // Set this delay time for how long each light should stay on.

    private bool isGreenLightActive;
    private int currentCountdown;

    private void Start()
    {
        StartCoroutine(ToggleLights());
    }

    private IEnumerator ToggleLights()
    {
        while (true)
        {
            // Activate green light and deactivate red light
            greenLight.enabled = true;
            redLight.enabled = false;
```

```
            isGreenLightActive = true;
            yield return StartCoroutine(Countdown());

            // Activate red light and deactivate green light
            greenLight.enabled = false;
            redLight.enabled = true;
            isGreenLightActive = false;
            yield return StartCoroutine(Countdown());
        }
    }

    private IEnumerator Countdown()
    {
        for (int i = (int)delay; i > 0; i--)
        {
            countdownText.text = i.ToString();
            currentCountdown = i;
            yield return new WaitForSeconds(1);
        }
    }

    public bool IsGreenLight()
    {
        return isGreenLightActive;
    }

    public int GetCurrentCountdown()
    {
        return currentCountdown;
    }
}
```

PinocchioMovement

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

public class PinocchioMovement : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(19, 2, 88);
    private Vector3 endPosition = new Vector3(-19, 2, 88);
    private bool isMoving = false;
    private float startTime;
    private float speed;

    public LightToggle lightToggle; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition
    public TextMeshProUGUI scoreText; // Assign this in the inspector

    void Start()
    {
        transform.position = startPosition;
        scoreText.text = "Score: 0/10"; // Initialize the score text at the start
        scoreText.gameObject.SetActive(true); // Show the score text at start
    }

    void Update()
    {
        if (isMoving)
```

```
        {
            float timeSinceStarted = Time.time - startTime;
            float fractionOfJourney = timeSinceStarted / speed;
            transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

            // Check if Pinocchio has reached the end position
            if (Vector3.Distance(transform.position, endPosition) < 0.01f)
            {
                isMoving = false;
            }
        }
    }

    public void StartWalking()
    {
        StartMoving(walkDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() >= 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
        }
        else
        {
```

```
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    public void StartRunning()
    {
        StartMoving(runDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() < 5)
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
            else if (lightToggle.GetCurrentCountdown() >= 5 &&
lightToggle.GetCurrentCountdown() < 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            // Modified condition to update the score to 0/10 when the counter is
between 15 and 10
            else if (lightToggle.GetCurrentCountdown() >= 10 &&
lightToggle.GetCurrentCountdown() <= 15)
            {
                UpdateScore(false); // Update score to 0/10
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
```

```
        }
        else
        {
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private void StartMoving(float duration)
    {
        if (!isMoving)
        {
            isMoving = true;
            startTime = Time.time;
            speed = 1.0f / duration;
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }
    private void UpdateScore(bool conditionMet)
    {
        if (conditionMet)
            scoreText.text = "Score: 10/10";
        else
            scoreText.text = "Score: 0/10";
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 7 – DESKTOP VRBCS II

ButtonsInteraction19

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;   // For Button
using TMPro;            // For TextMeshPro (TMP_Text)

public class ButtonsInteraction19 : MonoBehaviour
{
    [Header("Texts (TextMeshPro - assign in Inspector)")]
    public TMP_Text text1;
    public TMP_Text text2;
    public TMP_Text text3;
    public TMP_Text text4;

    [Header("N Text (TextMeshPro - assign in Inspector)")]
    public TMP_Text nText;   // This will show: "N = x"

    [Header("Buttons (UI Buttons - assign in Inspector)")]
    public Button button2;   // Starts 2 → 1 + cubes (2 & 3)
    public Button button3;   // Starts 3 → 2 → 1 + cubes (1–6, end with 4,5,6)
    public Button button4;   // Starts 4 → 3 → 2 → 1 + cubes (1–10)

    [Header("Timing")]
    [Tooltip("Time in seconds between each text appearing.")]
    public float delayBetweenTexts = 1f;

    [Header("Cubes (GameObjects - assign in Inspector)")]
    public GameObject cube1;
```

```
    public GameObject cube2;
    public GameObject cube3;
    public GameObject cube4;
    public GameObject cube5;
    public GameObject cube6;
    public GameObject cube7;
    public GameObject cube8;
    public GameObject cube9;
    public GameObject cube10;

    private bool sequenceStarted = false;

    void Start()
    {
        bool missing = false;

        // Texts
        if (text1 == null) { Debug.LogError("ButtonsInteraction19: text1 is NOT assigned!", this); missing = true; }
        if (text2 == null) { Debug.LogError("ButtonsInteraction19: text2 is NOT assigned!", this); missing = true; }
        if (text3 == null) { Debug.LogError("ButtonsInteraction19: text3 is NOT assigned!", this); missing = true; }
        if (text4 == null) { Debug.LogError("ButtonsInteraction19: text4 is NOT assigned!", this); missing = true; }

        // N text (not fatal if missing, but we warn)
        if (nText == null) { Debug.LogError("ButtonsInteraction19: nText is NOT assigned!", this); }

        // Buttons
        if (button2 == null) { Debug.LogError("ButtonsInteraction19: button2 is NOT assigned!", this); missing = true; }
```

```
    if (button3 == null) { Debug.LogError("ButtonsInteraction19: button3 is NOT assigned!", this); missing = true; }
    if (button4 == null) { Debug.LogError("ButtonsInteraction19: button4 is NOT assigned!", this); missing = true; }


    if (missing)
    {
        Debug.LogError("ButtonsInteraction19: Fix the missing references in the Inspector.", this);
        return;
    }


    // Clamp negative delay
    if (delayBetweenTexts < 0f)
    {
        Debug.LogWarning("ButtonsInteraction19: delayBetweenTexts was negative, clamping to 0.");
        delayBetweenTexts = 0f;
    }


    // Hide all texts at the start
    text1.gameObject.SetActive(false);
    text2.gameObject.SetActive(false);
    text3.gameObject.SetActive(false);
    text4.gameObject.SetActive(false);


    // Hide N text at the start
    if (nText != null)
    {
        nText.gameObject.SetActive(false);
    }
```

```
    // Hide all cubes at the start
    SetAllCubesActive(false);

    // Ensure buttons are visible
    button2.gameObject.SetActive(true);
    button3.gameObject.SetActive(true);
    button4.gameObject.SetActive(true);

    // Add listeners
    button2.onClick.AddListener(OnButton2Clicked);
    button3.onClick.AddListener(OnButton3Clicked);
    button4.onClick.AddListener(OnButton4Clicked);
}

// Helper to set "N = x" (ENGLISH N)
private void SetNValue(int n)
{
    if (nText != null)
    {
        nText.gameObject.SetActive(true);
        nText.text = "N = " + n.ToString();
    }
}

// BUTTON 4: 4 → 3 → 2 → 1, then full cube sequence (1–10)
private void OnButton4Clicked()
{
    if (sequenceStarted) return;
    SetNValue(4);
    sequenceStarted = true;
    StartCoroutine(ShowTextsSequence_From4());
```

```
}

// BUTTON 3: 3 → 2 → 1, then cube sequence ending with 4,5,6 visible
private void OnButton3Clicked()
{
    if (sequenceStarted) return;
    SetNValue(3);
    sequenceStarted = true;
    StartCoroutine(ShowTextsSequence_From3());
}

// BUTTON 2: 2 → 1, then cube sequence ending with 2 & 3 visible
private void OnButton2Clicked()
{
    if (sequenceStarted) return;
    SetNValue(2);
    sequenceStarted = true;
    StartCoroutine(ShowTextsSequence_From2());
}

private IEnumerator ShowTextsSequence_From4()
{
    yield return new WaitForSeconds(delayBetweenTexts);
    text4.gameObject.SetActive(true);

    yield return new WaitForSeconds(delayBetweenTexts);
    text3.gameObject.SetActive(true);

    yield return new WaitForSeconds(delayBetweenTexts);
    text2.gameObject.SetActive(true);
```

```
    yield return new WaitForSeconds(delayBetweenTexts);
    text1.gameObject.SetActive(true);

    // After all texts are done, start full cube sequence for Button 4
    StartCoroutine(CubeSequence_Full());
}

private IEnumerator ShowTextsSequence_From3()
{
    yield return new WaitForSeconds(delayBetweenTexts);
    text3.gameObject.SetActive(true);

    yield return new WaitForSeconds(delayBetweenTexts);
    text2.gameObject.SetActive(true);

    yield return new WaitForSeconds(delayBetweenTexts);
    text1.gameObject.SetActive(true);

    // After texts 3 → 2 → 1, start cube sequence (ending with 4,5,6 visible)
    StartCoroutine(CubeSequence_UpTo456());
}

private IEnumerator ShowTextsSequence_From2()
{
    yield return new WaitForSeconds(delayBetweenTexts);
    text2.gameObject.SetActive(true);

    yield return new WaitForSeconds(delayBetweenTexts);
    text1.gameObject.SetActive(true);

    // After texts 2 → 1, start cube sequence ending with 2 & 3 visible
```

```
    StartCoroutine(CubeSequence_UpTo23());
}

// --- CUBE SEQUENCE FOR BUTTON 4 (FULL: 1–10) ---
// uses fixed 2-second gaps
private IEnumerator CubeSequence_Full()
{
    float cubeDelay = 2f;

    // Start with everything off
    SetAllCubesActive(false);

    // 1) cube1 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(true);

    // 2) cube1 disappears, cube2 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(false);
    if (cube2 != null) cube2.SetActive(true);

    // 3) cube3 appears, cube2 stays visible
    yield return new WaitForSeconds(cubeDelay);
    if (cube3 != null) cube3.SetActive(true);

    // 4) cubes 2 and 3 disappear
    yield return new WaitForSeconds(cubeDelay);
    if (cube2 != null) cube2.SetActive(false);
    if (cube3 != null) cube3.SetActive(false);

    // 5) cubes 4 and 5 appear together
```

```
    yield return new WaitForSeconds(cubeDelay);
    if (cube4 != null) cube4.SetActive(true);
    if (cube5 != null) cube5.SetActive(true);

    // 6) cube6 appears (4 and 5 stay visible)
    yield return new WaitForSeconds(cubeDelay);
    if (cube6 != null) cube6.SetActive(true);

    // 7) cubes 4, 5, and 6 disappear
    yield return new WaitForSeconds(cubeDelay);
    if (cube4 != null) cube4.SetActive(false);
    if (cube5 != null) cube5.SetActive(false);
    if (cube6 != null) cube6.SetActive(false);

    // 8) cubes 7, 8, and 9 appear together
    yield return new WaitForSeconds(cubeDelay);
    if (cube7 != null) cube7.SetActive(true);
    if (cube8 != null) cube8.SetActive(true);
    if (cube9 != null) cube9.SetActive(true);

    // 9) cube10 appears (7, 8, 9 stay visible)
    yield return new WaitForSeconds(cubeDelay);
    if (cube10 != null) cube10.SetActive(true);
}

// --- CUBE SEQUENCE FOR BUTTON 3 (END WITH 4,5,6 VISIBLE) ---
private IEnumerator CubeSequence_UpTo456()
{
    float cubeDelay = 2f;

    // Start with everything off
```

```
    SetAllCubesActive(false);

    // 1) cube1 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(true);

    // 2) cube1 disappears, cube2 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(false);
    if (cube2 != null) cube2.SetActive(true);

    // 3) cube3 appears, cube2 stays visible
    yield return new WaitForSeconds(cubeDelay);
    if (cube3 != null) cube3.SetActive(true);

    // 4) cubes 2 and 3 disappear
    yield return new WaitForSeconds(cubeDelay);
    if (cube2 != null) cube2.SetActive(false);
    if (cube3 != null) cube3.SetActive(false);

    // 5) cubes 4 and 5 appear together
    yield return new WaitForSeconds(cubeDelay);
    if (cube4 != null) cube4.SetActive(true);
    if (cube5 != null) cube5.SetActive(true);

    // 6) cube6 appears (4 and 5 stay visible)
    yield return new WaitForSeconds(cubeDelay);
    if (cube6 != null) cube6.SetActive(true);
    // Final state for Button 3: cubes 4, 5, and 6 visible
}
```

```
// --- CUBE SEQUENCE FOR BUTTON 2 (END WITH 2 & 3 VISIBLE) ---
private IEnumerator CubeSequence_UpTo23()
{
    float cubeDelay = 2f;

    // Start with everything off
    SetAllCubesActive(false);

    // 1) cube1 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(true);

    // 2) cube1 disappears, cube2 appears
    yield return new WaitForSeconds(cubeDelay);
    if (cube1 != null) cube1.SetActive(false);
    if (cube2 != null) cube2.SetActive(true);

    // 3) cube3 appears, cube2 stays visible
    yield return new WaitForSeconds(cubeDelay);
    if (cube3 != null) cube3.SetActive(true);
    // Final state for Button 2: cubes 2 and 3 visible
}

private void SetAllCubesActive(bool active)
{
    if (cube1 != null) cube1.SetActive(active);
    if (cube2 != null) cube2.SetActive(active);
    if (cube3 != null) cube3.SetActive(active);
    if (cube4 != null) cube4.SetActive(active);
    if (cube5 != null) cube5.SetActive(active);
    if (cube6 != null) cube6.SetActive(active);
```

```
        if (cube7 != null) cube7.SetActive(active);
        if (cube8 != null) cube8.SetActive(active);
        if (cube9 != null) cube9.SetActive(active);
        if (cube10 != null) cube10.SetActive(active);
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 8 – DESKTOP VRBCS II

ButtonsInteraction20

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

public class ButtonsInteraction20 : MonoBehaviour
{
    [Header("Start Buttons")]
    public Button Start_1;
    public Button start_2;

    [Header("Answer Buttons")]
    public Button button_2;
    public Button button_3;
    public Button button_4;

    [Header("Score Text (TextMeshPro)")]
    public TMP_Text text_score;

    [Header("Instructions Texts (TextMeshPro)")]
    public TMP_Text text_instrustions_1;

    [Header("Phase 2 UI Texts (TextMeshPro)")]
    public TMP_Text text_instrustions_2;
    public TMP_Text answer_text_1;
    public TMP_Text answer_text_2;
    public TMP_Text answer_text_3;
```

```
[Header("Cubes")]
public GameObject cube1;
public GameObject cube2;
public GameObject cube3;
public GameObject cube4;
public GameObject cube5;
public GameObject cube6;

[Header("Timing")]
public float cubeDelay = 2f;
public float answerWindowSecondsPhase1 = 7f;
public float answerWindowSecondsPhase2 = 10f;

private bool sequenceStarted = false;      // Start_1 only
private bool phase2Started = false;        // start_2 only
private bool awaitingAnswer = false;
private bool answered = false;

private bool isPhase2AnswerWindow = false;
private bool phase2UIShown = false;

private int firstPhaseScore = 0;   // 0 or 10
private int totalScore = 0;        // 0, 10, or 20

private Coroutine answerTimerRoutine;

void Start()
{
    bool missing = false;
```

```
        if (Start_1 == null) { Debug.LogError("ButtonsInteraction20: Start_1 is NOT assigned!", this); missing = true; }

        if (start_2 == null) { Debug.LogError("ButtonsInteraction20: start_2 is NOT assigned!", this); missing = true; }


        if (button_2 == null) { Debug.LogError("ButtonsInteraction20: button_2 is NOT assigned!", this); missing = true; }

        if (button_3 == null) { Debug.LogError("ButtonsInteraction20: button_3 is NOT assigned!", this); missing = true; }

        if (button_4 == null) { Debug.LogError("ButtonsInteraction20: button_4 is NOT assigned!", this); missing = true; }


        if (text_score == null) { Debug.LogError("ButtonsInteraction20: text_score is NOT assigned!", this); missing = true; }


        if (text_instrustions_1 == null) { Debug.LogError("ButtonsInteraction20: text_instrustions_1 is NOT assigned!", this); missing = true; }


        if (text_instrustions_2 == null) { Debug.LogError("ButtonsInteraction20: text_instrustions_2 is NOT assigned!", this); missing = true; }

        if (answer_text_1 == null) { Debug.LogError("ButtonsInteraction20: answer_text_1 is NOT assigned!", this); missing = true; }

        if (answer_text_2 == null) { Debug.LogError("ButtonsInteraction20: answer_text_2 is NOT assigned!", this); missing = true; }

        if (answer_text_3 == null) { Debug.LogError("ButtonsInteraction20: answer_text_3 is NOT assigned!", this); missing = true; }


        if (cube1 == null) { Debug.LogError("ButtonsInteraction20: cube1 is NOT assigned!", this); missing = true; }

        if (cube2 == null) { Debug.LogError("ButtonsInteraction20: cube2 is NOT assigned!", this); missing = true; }

        if (cube3 == null) { Debug.LogError("ButtonsInteraction20: cube3 is NOT assigned!", this); missing = true; }

        if (cube4 == null) { Debug.LogError("ButtonsInteraction20: cube4 is NOT assigned!", this); missing = true; }
```

```
    if (cube5 == null) { Debug.LogError("ButtonsInteraction20: cube5 is NOT assigned!", this); missing = true; }
    if (cube6 == null) { Debug.LogError("ButtonsInteraction20: cube6 is NOT assigned!", this); missing = true; }


    if (missing)
    {
        Debug.LogError("ButtonsInteraction20: Fix missing references in the Inspector.", this);
        return;
    }


    if (cubeDelay < 0f) cubeDelay = 0f;
    if (answerWindowSecondsPhase1 < 0f) answerWindowSecondsPhase1 = 0f;
    if (answerWindowSecondsPhase2 < 0f) answerWindowSecondsPhase2 = 0f;


    SetAllCubesActive(false);


    // Show instructions_1 at the start of the scene (stays visible until user answers in Phase 1)
    text_instrustions_1.gameObject.SetActive(true);


    // Score text hidden at start (optional)
    text_score.gameObject.SetActive(false);


    // Phase 2 texts hidden at start
    text_instrustions_2.gameObject.SetActive(false);
    answer_text_1.gameObject.SetActive(false);
    answer_text_2.gameObject.SetActive(false);
    answer_text_3.gameObject.SetActive(false);


    // Disable answer buttons until a window opens
```

```
    SetAnswerButtonsInteractable(false);

    // start_2 locked until after phase 1 score is shown + 3 seconds
    start_2.interactable = false;

    Start_1.onClick.AddListener(OnStart1Clicked);
    start_2.onClick.AddListener(OnStart2Clicked);

    button_2.onClick.AddListener(OnButton2Clicked);
    button_3.onClick.AddListener(OnButton3Clicked);
    button_4.onClick.AddListener(OnButton4Clicked);
}

private void OnStart1Clicked()
{
    if (sequenceStarted) return;

    sequenceStarted = true;

    // Reset scoring state
    awaitingAnswer = false;
    answered = false;
    isPhase2AnswerWindow = false;

    firstPhaseScore = 0;
    totalScore = 0;

    phase2Started = false;
    phase2UIShown = false;

    if (answerTimerRoutine != null) StopCoroutine(answerTimerRoutine);
```

```
    text_score.gameObject.SetActive(false);

    // Keep phase 2 UI hidden until after score + 3 sec
    text_instrustions_2.gameObject.SetActive(false);
    answer_text_1.gameObject.SetActive(false);
    answer_text_2.gameObject.SetActive(false);
    answer_text_3.gameObject.SetActive(false);
    start_2.interactable = false;

    // Instruction 1 should be visible for Phase 1
    text_instrustions_1.gameObject.SetActive(true);

    SetAnswerButtonsInteractable(false);

    StartCoroutine(CubeSequence_UpTo456(false)); // phase 1
}

private void OnStart2Clicked()
{
    if (!phase2UIShown) return;
    if (phase2Started) return;

    phase2Started = true;

    awaitingAnswer = false;
    answered = false;
    isPhase2AnswerWindow = false;

    if (answerTimerRoutine != null) StopCoroutine(answerTimerRoutine);
```

```
        SetAnswerButtonsInteractable(false);

        StartCoroutine(CubeSequence_UpTo456(true)); // phase 2
    }

    // Same cube sequence as before; parameter decides which phase answer window to open
    private IEnumerator CubeSequence_UpTo456(bool phase2)
    {
        SetAllCubesActive(false);

        // 1) cube1 appears
        yield return new WaitForSeconds(cubeDelay);
        cube1.SetActive(true);

        // 2) cube1 disappears, cube2 appears
        yield return new WaitForSeconds(cubeDelay);
        cube1.SetActive(false);
        cube2.SetActive(true);

        // 3) cube3 appears, cube2 stays visible
        yield return new WaitForSeconds(cubeDelay);
        cube3.SetActive(true);

        // 4) cubes 2 and 3 disappear
        yield return new WaitForSeconds(cubeDelay);
        cube2.SetActive(false);
        cube3.SetActive(false);

        // 5) cubes 4 and 5 appear together
        yield return new WaitForSeconds(cubeDelay);
```

```
        cube4.SetActive(true);
        cube5.SetActive(true);

        // 6) cube6 appears (4 and 5 stay visible)
        yield return new WaitForSeconds(cubeDelay);
        cube6.SetActive(true);

        // Open answer window (Phase 1: 7s, Phase 2: 10s)
        BeginAnswerWindow(phase2);
    }

    private void BeginAnswerWindow(bool phase2)
    {
        awaitingAnswer = true;
        answered = false;
        isPhase2AnswerWindow = phase2;

        SetAnswerButtonsInteractable(true);

        if (answerTimerRoutine != null) StopCoroutine(answerTimerRoutine);
        float seconds = phase2 ? answerWindowSecondsPhase2 :
answerWindowSecondsPhase1;
        answerTimerRoutine = StartCoroutine(AnswerWindowTimer(seconds));
    }

    private IEnumerator AnswerWindowTimer(float seconds)
    {
        yield return new WaitForSeconds(seconds);

        if (!answered)
        {
```

```
            awaitingAnswer = false;
            SetAnswerButtonsInteractable(false);

            // Phase 1 timeout counts as 0/20 (same as before)
            if (!isPhase2AnswerWindow)
            {
                // End of Phase 1 -> hide instruction 1
                if (text_instrustions_1 != null)
text_instrustions_1.gameObject.SetActive(false);

                firstPhaseScore = 0;
                totalScore = firstPhaseScore;
                SetScore("Score: " + totalScore + "/20");

                if (!phase2UIShown)
                    StartCoroutine(ShowPhase2UIAfterDelay());
            }
            // Phase 2 timeout: score remains the same as in first phase
        }
    }

    private void OnButton3Clicked()
    {
        if (!awaitingAnswer || answered) return;

        answered = true;
        awaitingAnswer = false;

        SetAnswerButtonsInteractable(false);

        if (!isPhase2AnswerWindow)
```

```
        {
            // End of Phase 1 -> hide instruction 1
            if (text_instrustions_1 != null)
text_instrustions_1.gameObject.SetActive(false);

            // Phase 1 correct (unchanged)
            firstPhaseScore = 10;
            totalScore = firstPhaseScore;
            SetScore("Score: " + totalScore + "/20");

            if (!phase2UIShown)
                StartCoroutine(ShowPhase2UIAfterDelay());
        }
        else
        {
            // Phase 2: button_3 is now WRONG -> score remains the same as in first
phase
            SetScore("Score: " + totalScore + "/20");
        }
    }

    private void OnButton2Clicked()
    {
        if (!awaitingAnswer || answered) return;

        answered = true;
        awaitingAnswer = false;

        SetAnswerButtonsInteractable(false);

        if (!isPhase2AnswerWindow)
```

```
        {
            // End of Phase 1 -> hide instruction 1
            if (text_instrustions_1 != null)
text_instrustions_1.gameObject.SetActive(false);

            // Phase 1 wrong (unchanged)
            firstPhaseScore = 0;
            totalScore = firstPhaseScore;
            SetScore("Score: " + totalScore + "/20");

            if (!phase2UIShown)
                StartCoroutine(ShowPhase2UIAfterDelay());
        }
        else
        {
            // Phase 2: button_2 is now CORRECT -> add +10 to first phase score (0->10,
10->20)
            totalScore = Mathf.Min(firstPhaseScore + 10, 20);
            SetScore("Score: " + totalScore + "/20");
        }
    }

    private void OnButton4Clicked()
    {
        if (!awaitingAnswer || answered) return;

        answered = true;
        awaitingAnswer = false;

        SetAnswerButtonsInteractable(false);
```

```
        if (!isPhase2AnswerWindow)
        {
            // End of Phase 1 -> hide instruction 1
            if (text_instrustions_1 != null)
text_instrustions_1.gameObject.SetActive(false);

            // Phase 1 wrong (unchanged)
            firstPhaseScore = 0;
            totalScore = firstPhaseScore;
            SetScore("Score: " + totalScore + "/20");

            if (!phase2UIShown)
                StartCoroutine(ShowPhase2UIAfterDelay());
        }
        else
        {
            // Phase 2 wrong -> score remains the same as in first phase
            SetScore("Score: " + totalScore + "/20");
        }
    }

    private void SetScore(string value)
    {
        text_score.gameObject.SetActive(true);
        text_score.text = value;
    }

    private IEnumerator ShowPhase2UIAfterDelay()
    {
        phase2UIShown = true;
```

```
        // After the score appeared, wait 3 sec then show Phase 2 UI texts
        yield return new WaitForSeconds(3f);

        text_instrustions_2.gameObject.SetActive(true);
        answer_text_1.gameObject.SetActive(true);
        answer_text_2.gameObject.SetActive(true);
        answer_text_3.gameObject.SetActive(true);

        // Unlock start_2
        start_2.interactable = true;
    }

    private void SetAnswerButtonsInteractable(bool on)
    {
        button_2.interactable = on;
        button_3.interactable = on;
        button_4.interactable = on;
    }

    private void SetAllCubesActive(bool active)
    {
        cube1.SetActive(active);
        cube2.SetActive(active);
        cube3.SetActive(active);
        cube4.SetActive(active);
        cube5.SetActive(active);
        cube6.SetActive(active);
    }
}
```

IMMERSIVE VR VRBCS 1

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 1

MoveCubeAround

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

// Include the UI namespace to use Text

public class MoveCubeAround : MonoBehaviour
{
   public GameObject Cube_Button1;  // Button to move the cube forward
   public GameObject Cube_Button2;  // Button to move the cube right
   public GameObject Cube_Button3;  // Button to move the cube backward
   public GameObject Cube_Button4;  // Button to move the cube left
   public GameObject Cube_Button5;  // Run button to confirm and move the cube

   // Buttons for specifying movement magnitude
   public GameObject Cube_ButtonN1;
   public GameObject Cube_ButtonN2;
   public GameObject Cube_ButtonN3;
   public GameObject Cube_ButtonN4;
   public GameObject Cube_ButtonN5;
   public GameObject Cube_ButtonN6;
   public GameObject Cube_ButtonN7;
   public GameObject Cube_ButtonN8;

   public MoveAround targetScript;  // Script that contains the movement logic

   public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

   private bool readyToMoveForward = false;
   private bool readyToMoveBackward = false;
   private bool readyToMoveRight = false;
   private bool readyToMoveLeft = false;
   private int moveMagnitude = 0;  // To store the movement magnitude

   void OnCollisionEnter(Collision collisionInfo)
   {
      // Handle directional buttons
```

```
        if (collisionInfo.collider.name == "Cube_Button1")
            readyToMoveForward = true;
        else if (collisionInfo.collider.name == "Cube_Button2")
            readyToMoveRight = true;
        else if (collisionInfo.collider.name == "Cube_Button3")
            readyToMoveBackward = true;
        else if (collisionInfo.collider.name == "Cube_Button4")
            readyToMoveLeft = true;

        // Handle magnitude buttons
        if (collisionInfo.collider.name.StartsWith("Cube_ButtonN"))
        {
            moveMagnitude = int.Parse(collisionInfo.collider.name.Substring(12));  //
Extracting the number from the name
        }

        // Run button
        if (collisionInfo.collider.name == "Cube_Button5" && moveMagnitude > 0)
        {
            if (readyToMoveForward)
            {
                targetScript.MoveCube("forward", moveMagnitude);
                actionText.text = $"MoveForward({moveMagnitude})";
            }
            else if (readyToMoveBackward)
            {
                targetScript.MoveCube("backward", moveMagnitude);
                actionText.text = $"MoveBackward({moveMagnitude})";
            }
            else if (readyToMoveRight)
            {
                targetScript.MoveCube("right", moveMagnitude);
                actionText.text = $"MoveRight({moveMagnitude})";
            }
            else if (readyToMoveLeft)
            {
                targetScript.MoveCube("left", moveMagnitude);
                actionText.text = $"MoveLeft({moveMagnitude})";
            }

            ResetReadyStates();
        }
    }

    private void ResetReadyStates()
    {
        readyToMoveForward = false;
```

```
        readyToMoveBackward = false;
        readyToMoveRight = false;
        readyToMoveLeft = false;
        moveMagnitude = 0;  // Reset magnitude
    }
}
```

MoveAround

```
using UnityEngine;

public class MoveAround : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector

   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };

   void Start()
   {
      // Set the cube's starting position
      cube.transform.position = new Vector3(10f, 0.08f, 10f);
      Debug.Log("Cube Start Position: " + cube.transform.position);
   }

   public void MoveCube(string direction, int magnitude)
   {
      Vector3 proposedMove = Vector3.zero;

      switch (direction)
      {
         case "forward":
            proposedMove = Vector3.right * magnitude; // Moves in +X direction
            break;
         case "backward":
            proposedMove = Vector3.left * magnitude; // Moves in -X direction
            break;
         case "right":
            proposedMove = Vector3.back * magnitude; // Moves in -Z direction
            break;
         case "left":
            proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
            break;
      }
```

```
        Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");

        if (CanMove(proposedMove))
        {
            cube.transform.position += proposedMove;
            Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
        }
        else
        {
            Debug.Log($"Move blocked: {direction} at position {cube.transform.position +
proposedMove}");
        }
    }

    private bool CanMove(Vector3 proposedMove)
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point
precision
            {
                return true;
            }
        }
        return false; // Block move if it does not exactly match any allowed position
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 2

MoveAround_2

```
using UnityEngine;

public class MoveAround_2 : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector
   //VR

   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };

   void Start()
   {
      // Set the cube's starting position and rotation
      cube.transform.position = new Vector3(10f, 0.012f, 10f);
      cube.transform.rotation = Quaternion.Euler(0, 90, 0);

      Debug.Log("Cube Start Position: " + cube.transform.position);
      Debug.Log("Cube Start Rotation: " + cube.transform.rotation.eulerAngles);
   }

   public void MoveCube(string direction, int magnitude)
   {
      Vector3 proposedMove = Vector3.zero;

      switch (direction)
      {
         case "forward":
            proposedMove = Vector3.right * magnitude; // Moves in +X direction
            break;
         case "backward":
            proposedMove = Vector3.left * magnitude; // Moves in -X direction
            break;
         case "right":
```

```
            proposedMove = Vector3.back * magnitude; // Moves in -Z direction
            break;
        case "left":
            proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
            break;
    }

    Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");

    if (CanMove(proposedMove))
    {
        cube.transform.position += proposedMove;
        Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
    }
    else
    {
        Debug.Log($"Move blocked: {direction} at position {cube.transform.position +
proposedMove}");
    }
  }

  private bool CanMove(Vector3 proposedMove)
  {
    Vector3 newPosition = cube.transform.position + proposedMove;
    foreach (Vector3 pos in allowedPositions)
    {
        if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point
precision
        {
            return true;
        }
    }
    return false; // Block move if it does not exactly match any allowed position
  }
}
```

MoveCubeAround_2

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

public class MoveCubeAround_2 : MonoBehaviour
{
   public GameObject Cube_Button1;  // Button to move the cube forward
   public GameObject Cube_Button2;  // Button to move the cube right
   public GameObject Cube_Button3;  // Button to move the cube backward
   public GameObject Cube_Button4;  // Button to move the cube left
   public GameObject Cube_Button5;  // Run button to confirm and move the cube

   public GameObject Cube_ButtonN1;
   public GameObject Cube_ButtonN2;
   public GameObject Cube_ButtonN3;
   public GameObject Cube_ButtonN4;
   public GameObject Cube_ButtonN5;
   public GameObject Cube_ButtonN6;
   public GameObject Cube_ButtonN7;
   public GameObject Cube_ButtonN8;

   public MoveAround_2 targetScript;  // Script that contains the movement logic

   public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

   private bool readyToMoveForward = false;
   private bool readyToMoveBackward = false;
   private bool readyToMoveRight = false;
   private bool readyToMoveLeft = false;
   private int moveMagnitude = 0;  // To store the movement magnitude

   void OnCollisionEnter(Collision collisionInfo)
   {
      // Handle directional buttons
      if (collisionInfo.collider.name == "Cube_Button1")
      {
         readyToMoveForward = true;
         targetScript.cube.transform.rotation = Quaternion.Euler(0, 90, 0);
      }
      else if (collisionInfo.collider.name == "Cube_Button2")
      {
         readyToMoveRight = true;
         targetScript.cube.transform.rotation = Quaternion.Euler(0, 180, 0);
      }
```

```
        else if (collisionInfo.collider.name == "Cube_Button3")
        {
            readyToMoveBackward = true;
            targetScript.cube.transform.rotation = Quaternion.Euler(0, 270, 0);
        }
        else if (collisionInfo.collider.name == "Cube_Button4")
        {
            readyToMoveLeft = true;
            targetScript.cube.transform.rotation = Quaternion.Euler(0, 0, 0);
        }

        // Handle magnitude buttons
        if (collisionInfo.collider.name.StartsWith("Cube_ButtonN"))
        {
            moveMagnitude = int.Parse(collisionInfo.collider.name.Substring(12));  //
Extracting the number from the name
        }

        // Run button
        if (collisionInfo.collider.name == "Cube_Button5" && moveMagnitude > 0)
        {
            StartCoroutine(DelayedMove());
        }
    }

    private IEnumerator DelayedMove()
    {
        yield return new WaitForSeconds(3);  // Delay of 3 seconds

        if (readyToMoveForward)
        {
            targetScript.MoveCube("forward", moveMagnitude);
            actionText.text = $"MoveForward({moveMagnitude})";
        }
        else if (readyToMoveBackward)
        {
            targetScript.MoveCube("backward", moveMagnitude);
            actionText.text = $"MoveBackward({moveMagnitude})";
        }
        else if (readyToMoveRight)
        {
            targetScript.MoveCube("right", moveMagnitude);
            actionText.text = $"MoveRight({moveMagnitude})";
        }
        else if (readyToMoveLeft)
        {
            targetScript.MoveCube("left", moveMagnitude);
```

```
            actionText.text = $"MoveLeft({moveMagnitude})";
        }

        ResetReadyStates();
    }

    private void ResetReadyStates()
    {
        readyToMoveForward = false;
        readyToMoveBackward = false;
        readyToMoveRight = false;
        readyToMoveLeft = false;
        moveMagnitude = 0;  // Reset magnitude
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 3

MoveAround_2_CHANGED

```
using UnityEngine;

public class MoveAround_2_CHANGED : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector
   //VR

   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };

   void Start()
   {
      // Set the cube's starting position and rotation
      cube.transform.position = new Vector3(10f, 0.012f, 10f);
      cube.transform.rotation = Quaternion.Euler(0, 90, 0);

      Debug.Log("Cube Start Position: " + cube.transform.position);
      Debug.Log("Cube Start Rotation: " + cube.transform.rotation.eulerAngles);
   }

   public void MoveCube(string direction, int magnitude)
   {
      Vector3 proposedMove = Vector3.zero;

      switch (direction)
      {
         case "forward":
            proposedMove = Vector3.right * magnitude; // Moves in +X direction
            break;
         case "backward":
            proposedMove = Vector3.left * magnitude; // Moves in -X direction
            break;
         case "right":
```

```
            proposedMove = Vector3.back * magnitude; // Moves in -Z direction
            break;
        case "left":
            proposedMove = Vector3.forward * magnitude; // Moves in +Z direction
            break;
    }

    Debug.Log($"Attempting to move: {direction} by {magnitude}. Proposed move:
{proposedMove}");

    if (CanMove(proposedMove))
    {
        cube.transform.position += proposedMove;
        Debug.Log($"Moving: {direction} to new position {cube.transform.position}");
    }
    else
    {
        Debug.Log($"Move blocked: {direction} at position {cube.transform.position +
proposedMove}");
    }
  }

  private bool CanMove(Vector3 proposedMove)
  {
    Vector3 newPosition = cube.transform.position + proposedMove;
    foreach (Vector3 pos in allowedPositions)
    {
        if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point
precision
        {
            return true;
        }
    }
    return false; // Block move if it does not exactly match any allowed position
  }
}
```

MoveCubeAround_2_CHANGED

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

public class MoveCubeAround_2_CHANGED : MonoBehaviour
{
   public GameObject Cube_Button1;  // Button to move the cube forward
   public GameObject Cube_Button2;  // Button to move the cube right
   public GameObject Cube_Button3;  // Button to move the cube backward
   public GameObject Cube_Button4;  // Button to move the cube left
   public GameObject Cube_Button5;  // Run button to confirm and move the cube

   public GameObject Cube_ButtonN1;
   public GameObject Cube_ButtonN2;
   public GameObject Cube_ButtonN3;
   public GameObject Cube_ButtonN4;
   public GameObject Cube_ButtonN5;
   public GameObject Cube_ButtonN6;
   public GameObject Cube_ButtonN7;
   public GameObject Cube_ButtonN8;

   public MoveAround_2_CHANGED targetScript;  // Script that contains the
movement logic

   public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

   private bool readyToMoveForward = false;
   private bool readyToMoveBackward = false;
   private bool readyToMoveRight = false;
   private bool readyToMoveLeft = false;
   private int moveMagnitude = 0;  // To store the movement magnitude

   void OnCollisionEnter(Collision collisionInfo)
   {
      // Handle directional buttons
      if (collisionInfo.collider.name == "Cube_Button1")
      {
         readyToMoveForward = true;
         targetScript.cube.transform.rotation = Quaternion.Euler(0, 90, 0);
      }
      else if (collisionInfo.collider.name == "Cube_Button2")
      {
         readyToMoveRight = true;
         targetScript.cube.transform.rotation = Quaternion.Euler(0, 180, 0);
```

```
        }
        else if (collisionInfo.collider.name == "Cube_Button3")
        {
            readyToMoveBackward = true;
            targetScript.cube.transform.rotation = Quaternion.Euler(0, 270, 0);
        }
        else if (collisionInfo.collider.name == "Cube_Button4")
        {
            readyToMoveLeft = true;
            targetScript.cube.transform.rotation = Quaternion.Euler(0, 0, 0);
        }

        // Handle magnitude buttons
        if (collisionInfo.collider.name.StartsWith("Cube_ButtonN"))
        {
            moveMagnitude = int.Parse(collisionInfo.collider.name.Substring(12));  //
Extracting the number from the name
        }

        // Run button
        if (collisionInfo.collider.name == "Cube_Button5" && moveMagnitude > 0)
        {
            StartCoroutine(DelayedMove());
        }
    }

    private IEnumerator DelayedMove()
    {
        yield return new WaitForSeconds(3);  // Delay of 3 seconds

        if (readyToMoveForward)
        {
            targetScript.MoveCube("forward", moveMagnitude);
            actionText.text = $"Move(Forward, {moveMagnitude})";
        }
        else if (readyToMoveBackward)
        {
            targetScript.MoveCube("backward", moveMagnitude);
            actionText.text = $"Move(Backward, {moveMagnitude})";
        }
        else if (readyToMoveRight)
        {
            targetScript.MoveCube("right", moveMagnitude);
            actionText.text = $"Move(Right, {moveMagnitude})";
        }
        else if (readyToMoveLeft)
        {
```

```
            targetScript.MoveCube("left", moveMagnitude);
            actionText.text = $"Move(Left, {moveMagnitude})";
        }

        ResetReadyStates();
    }

    private void ResetReadyStates()
    {
        readyToMoveForward = false;
        readyToMoveBackward = false;
        readyToMoveRight = false;
        readyToMoveLeft = false;
        moveMagnitude = 0;  // Reset magnitude
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 4

MoveAround8

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

public class MoveAround8 : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector

   // List of explicitly allowed positions
   private List<Vector3> allowedPositions = new List<Vector3>
   {
      new Vector3(10, 0.08f, 10), // Origin (O)

      new Vector3(10.5f, 0.08f, 10), // N1
      new Vector3(11, 0.08f, 10), // N2
      new Vector3(11.5f, 0.08f, 10), // N3
      new Vector3(12, 0.08f, 10), // N4

      new Vector3(9.5f, 0.08f, 10),  // S1
      new Vector3(9, 0.08f, 10),  // S2
      new Vector3(8.5f, 0.08f, 10),  // S3
      new Vector3(8, 0.08f, 10),  // S4

      new Vector3(10, 0.08f, 9.5f), // E1
      new Vector3(10, 0.08f, 9), // E2
      new Vector3(10, 0.08f, 8.5f), // E3
      new Vector3(10, 0.08f, 8), // E4

      new Vector3(10, 0.08f, 10.5f),  // W1
      new Vector3(10, 0.08f, 11),  // W2
      new Vector3(10, 0.08f, 11.5f),  // W3
      new Vector3(10, 0.08f, 12),  // W4

      new Vector3(10.5f, 0.08f, 9.5f), // NE1
      new Vector3(11, 0.08f, 9), // NE2
      new Vector3(11.5f, 0.08f, 8.5f), // NE3
      new Vector3(12, 0.08f, 8), // NE4

      new Vector3(9.5f, 0.08f, 9.5f),  // SE1
      new Vector3(9, 0.08f, 9),  // SE2
      new Vector3(8.5f, 0.08f, 8.5f),  // SE3
      new Vector3(8, 0.08f, 8),  // SE4
```

```
        new Vector3(9.5f, 0.08f, 10.5f),   // SW1
        new Vector3(9, 0.08f, 11),   // SW2
        new Vector3(8.5f, 0.08f, 11.5f),   // SW3
        new Vector3(8, 0.08f, 12),   // SW4

        new Vector3(10.5f, 0.08f, 10.5f),  // NW1
        new Vector3(11, 0.08f, 11),   // NW2
        new Vector3(11.5f, 0.08f, 11.5f),   // NW3
        new Vector3(12, 0.08f, 12),   // NW4
    };

    // Dictionary of movement restrictions from specific positions
    private Dictionary<Vector3, List<Vector3>> movementRestrictions = new
Dictionary<Vector3, List<Vector3>>
   {
        {new Vector3(10.5f, 0.08f, 10), new List<Vector3>{ // N1
         new Vector3(10.5f, 0.08f, 9.5f),   //ne1
         new Vector3(10, 0.08f, 9.5f), //e1
         new Vector3(9.5f, 0.08f, 9.5f), //se1
         new Vector3(9.5f, 0.08f, 10.5f), // sw1
         new Vector3(10, 0.08f, 10.5f), //w1
         new Vector3(10.5f, 0.08f, 10.5f) //nw1


        }},
        {new Vector3(10.5f, 0.08f, 9.5f), new List<Vector3>{ // NE1
           new Vector3(10.5f, 0.08f, 10),  //n1
           new Vector3(10, 0.08f, 10), //e1
           new Vector3(9.5f, 0.08f, 9.5f), //se1
           new Vector3(9.5f, 0.08f, 10), //s1
           new Vector3(10, 0.08f, 10.5f), //w1
           new Vector3(10.5f, 0.08f, 10.5f), //nw1
           new Vector3(11, 0.08f, 10), //n2
           new Vector3(10, 0.08f, 9) //e2

        }},

        {new Vector3(10, 0.08f, 9.5f), new List<Vector3>{ // E1

           new Vector3(10.5f, 0.08f, 10), //n1
           new Vector3(10.5f, 0.08f, 9.5f), //ne1
           new Vector3(9.5f, 0.08f, 9.5f), //se1
           new Vector3(9.5f, 0.08f, 10), //s1
           new Vector3(9.5f, 0.08f, 10.5f), //sw1
           new Vector3(10.5f, 0.08f, 10.5f) //nw1
        }},
```

```
{new Vector3(9.5f, 0.08f, 9.5f), new List<Vector3>{ // SE1

   new Vector3(9, 0.08f, 10),//s1
   new Vector3(9.5f, 0.08f, 10), //sw1
    new Vector3(10, 0.08f, 10.5f), //w1
   new Vector3(10.5f, 0.08f, 10), //n1
   new Vector3(10.5f, 0.08f, 9.5f), //ne1
   new Vector3(10, 0.08f, 9.5f), //e1
   new Vector3(10, 0.08f, 9), //e2
   new Vector3(9, 0.08f, 10) //s2
}},


{new Vector3(9.5f, 0.08f, 10), new List<Vector3>{ //S1
   new Vector3(9.5f, 0.08f, 10.5f),  //sw1
   new Vector3(10, 0.08f, 11), //w1
   new Vector3(10, 0.08f, 10.5f), // nw1
   new Vector3(10.5f, 0.08f, 9.5f),// ne1
   new Vector3(10, 0.08f, 9.5f), //e1
   new Vector3(9.5f, 0.08f, 9.5f) //se1
}},


{new Vector3(9.5f, 0.08f, 10.5f), new List<Vector3>{ //SW1
   new Vector3(10, 0.08f, 10.5f),//w1
   new Vector3(10.5f, 0.08f, 10.5f), //nw1
   new Vector3(10.5f, 0.08f, 10), //n1
   new Vector3(10, 0.08f, 9.5f), // e1
   new Vector3(9.5f, 0.08f, 9.5f), //se1
   new Vector3(9.5f, 0.08f, 10), //s1
   new Vector3(9, 0.08f, 10), // s2
   new Vector3(10, 0.08f, 1) //w2
}},


{new Vector3(10, 0.08f, 10.5f), new List<Vector3>{ //W1
   new Vector3(10.5f, 0.08f, 10.5f),   //nw1
   new Vector3(10.5f, 0.08f, 10),  //n1
   new Vector3(10.5f, 0.08f, 9.5f),   //ne1
   new Vector3(9.5f, 0.08f, 9.5f),  //se1
   new Vector3(9.5f, 0.08f, 10),  //s1
   new Vector3(9.5f, 0.08f, 10.5f)  //sw1
}},

{new Vector3(10.5f, 0.08f, 10.5f), new List<Vector3>{ //NW1
   new Vector3(11, 0.08f, 10),  // n1
```

```
            new Vector3(10.5f, 0.08f, 10),   //e1
            new Vector3(9.5f, 0.08f, 10),  // s1
            new Vector3(9.5f, 0.08f, 10.5f),  // sw1
            new Vector3(10, 0.08f, 10.5f),  //w1
            new Vector3(10.5f, 0.08f, 9.5f),   // ne1
            new Vector3(10, 0.08f, 11),  // w2
            new Vector3(11, 0.08f, 10)   //n2
        }},


        {new Vector3(11, 0.08f, 10), new List<Vector3>{ //N2
        new Vector3(10.5f, 0.08f, 10.5f), //nw1
        new Vector3(10.5f, 0.08f, 9.5f), //ne1
         }},

        {new Vector3(8, 0.08f, 10), new List<Vector3>{ //S2
        new Vector3(9.5f, 0.08f, 10.5f), //sw1
        new Vector3(9.5f, 0.08f, 9.5f), //se1
        }},

        {new Vector3(10, 0.08f, 11), new List<Vector3>{ //W2
        new Vector3(10.5f, 0.08f, 10.5f), //nw1
        new Vector3(9.5f, 0.08f, 10.5f),// sw1
        }},

        {new Vector3(10, 0.08f, 9), new List<Vector3>{ //E2
        new Vector3(10.5f, 0.08f, 9.5f), //ne1
        new Vector3(9.5f, 0.08f, 9.5f), //se1
        }}


    };

    void Start()
    {
        cube.transform.position = new Vector3(10f, 0.08f, 10f); // Set the cube's starting
position
        cube.transform.rotation = Quaternion.Euler(0, 270, 0); // Set the cube's starting
rotation
    }

    public void MoveCube(Vector3 newPosition)
    {
        if (IsAllowedPosition(newPosition) && CanMove(cube.transform.position,
newPosition))
        {
            Vector3 direction = newPosition - cube.transform.position;
```

```
            Quaternion targetRotation = GetTargetRotation(direction);
            StartCoroutine(RotateAndMove(cube, newPosition, targetRotation));
        }
        else
        {
            Debug.Log($"Move blocked to position {newPosition}. Not an allowed position
or restricted move.");
        }
    }

    private bool IsAllowedPosition(Vector3 newPosition)
    {
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f)
            {
                return true;
            }
        }
        return false; // If not listed, it's not an allowed position
    }

    private bool CanMove(Vector3 currentPos, Vector3 newPosition)
    {
        if (movementRestrictions.ContainsKey(currentPos))
        {
            foreach (Vector3 restrictedPos in movementRestrictions[currentPos])
            {
                if (Vector3.Distance(newPosition, restrictedPos) < 0.09f)
                {
                    return false;
                }
            }
        }
        return true; // If no restrictions are found, allow the move
    }

    private Quaternion GetTargetRotation(Vector3 direction)
    {
        float deltaX = direction.x;
        float deltaZ = direction.z;
        if (Mathf.Abs(deltaZ) > Mathf.Abs(deltaX))
        {
            return deltaZ > 0 ? Quaternion.Euler(0, 0, 0) : Quaternion.Euler(0, 180, 0); //
North or South
        }
        else if (Mathf.Abs(deltaX) > Mathf.Abs(deltaZ))
```

```
    {
        return deltaX > 0 ? Quaternion.Euler(0, 90, 0) : Quaternion.Euler(0, 270, 0); // East or West
    }
    else
    {
        if (deltaX > 0 && deltaZ > 0) return Quaternion.Euler(0, 45, 0); // NE
        else if (deltaX > 0 && deltaZ < 0) return Quaternion.Euler(0, 135, 0); // SE
        else if (deltaX < 0 && deltaZ < 0) return Quaternion.Euler(0, 225, 0); // SW
        else return Quaternion.Euler(0, 315, 0); // NW
    }
  }

  private IEnumerator RotateAndMove(GameObject cube, Vector3 targetPosition, Quaternion targetRotation)
  {
    yield return new WaitForSeconds(3); // Three-second delay before moving

    float rotationDuration = 0.3f; // Time to rotate
    float moveDuration = 0.8f; // Time to move
    float elapsedTime = 0.0f;
    Quaternion startRotation = cube.transform.rotation;
    Vector3 startPosition = cube.transform.position;

    // Rotate
    while (elapsedTime < rotationDuration)
    {
        cube.transform.rotation = Quaternion.Lerp(startRotation, targetRotation, elapsedTime / rotationDuration);
        elapsedTime += Time.deltaTime;
        yield return null;
    }
    cube.transform.rotation = targetRotation; // Ensure final rotation

    // Reset elapsed time for movement
    elapsedTime = 0.0f;

    // Move
    while (elapsedTime < moveDuration)
    {
        cube.transform.position = Vector3.Lerp(startPosition, targetPosition, elapsedTime / moveDuration);
        elapsedTime += Time.deltaTime;
        yield return null;
    }
    cube.transform.position = targetPosition; // Ensure final position
  }
```

```
}
```

MoveCubeAround8

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

public class MoveCubeAround8 : MonoBehaviour
{
   public GameObject North; //Button_1
   public GameObject East; //Button_2
   public GameObject South; //Button_3
   public GameObject West; //Button_4
   public GameObject NE; //Button_5
   public GameObject SE; //Button_6
   public GameObject NW; //Button_7
   public GameObject SW; //Button_8
   public GameObject RUN; //RUN BUTTON

   public GameObject Button_N1;
   public GameObject Button_N2;
   public GameObject Button_N3;
   public GameObject Button_N4;
   public GameObject Button_N5;
   public GameObject Button_N6;
   public GameObject Button_N7;
   public GameObject Button_N8;
   public MoveAround8 targetScript;  // Reference to the script that contains the
movement logic
   public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

   private string currentDirection = "";
   private int moveMagnitude = 0;
   private string displayText = "";

   private void OnCollisionEnter(Collision collisionInfo)
   {
      if (collisionInfo.collider.name == "Cube_North")
         SetDirection("right", "MoveNorth");
      else if (collisionInfo.collider.name == "Cube_East")
         SetDirection("backward", "MoveEast");
      else if (collisionInfo.collider.name == "Cube_South")
         SetDirection("left", "MoveSouth");
      else if (collisionInfo.collider.name == "Cube_West")
         SetDirection("forward", "MoveWest");
      else if (collisionInfo.collider.name == "Cube_NE")
```

```
            SetDirection("SE", "MoveNE");
        else if (collisionInfo.collider.name == "Cube_SE")
            SetDirection("SW", "MoveSE");
        else if (collisionInfo.collider.name == "Cube_SW")
            SetDirection("NW", "MoveSW");
        else if (collisionInfo.collider.name == "Cube_NW")
            SetDirection("NE", "MoveNW");
        else if (collisionInfo.collider.name == "Cube_RUN")
            StartCoroutine(DelayedProcessMove());
        else if (collisionInfo.collider.name.StartsWith("Cube_ButtonN"))
        {
            int magnitude = int.Parse(collisionInfo.collider.name.Replace("Cube_ButtonN",
"""));
            SetMagnitude(magnitude);
        }
    }

    private void SetDirection(string direction, string displayText)
    {
        currentDirection = direction;
        this.displayText = displayText;
        Debug.Log($"Direction set to {currentDirection}");
    }

    private void SetMagnitude(int magnitude)
    {
        moveMagnitude = magnitude;
        Debug.Log($"Magnitude set to {moveMagnitude}");
    }

    private IEnumerator DelayedProcessMove()
    {
        yield return new WaitForSeconds(3); // Wait for 3 seconds
        ProcessMove();
    }

    private void ProcessMove()
    {
        if (moveMagnitude > 0 && !string.IsNullOrEmpty(currentDirection))
        {
            Vector3 initialPosition = targetScript.cube.transform.position;
            Vector3 targetPosition = CalculateTargetPosition(currentDirection,
moveMagnitude);
            targetScript.MoveCube(targetPosition);

            if (targetScript.cube.transform.position != initialPosition)
            {
```

```
            UpdateActionText($"{displayText} ({moveMagnitude})");
        }
        else
        {
            UpdateActionText($"{displayText} ({moveMagnitude})");
        }
    }
    else
    {
        UpdateActionText("Move command ignored: No direction or magnitude set.");
    }
}

private Vector3 CalculateTargetPosition(string direction, int magnitude)
{
    float halfMagnitude = magnitude / 2.0f; // Halving the magnitude for movement
    Vector3 currentPosition = targetScript.cube.transform.position;
    switch (direction)
    {
        case "forward":
            return currentPosition + new Vector3(0, 0, halfMagnitude);
        case "right":
            return currentPosition + new Vector3(halfMagnitude, 0, 0);
        case "backward":
            return currentPosition - new Vector3(0, 0, halfMagnitude);
        case "left":
            return currentPosition - new Vector3(halfMagnitude, 0, 0);
        case "NE":
            return currentPosition + new Vector3(halfMagnitude, 0, halfMagnitude);
        case "SE":
            return currentPosition + new Vector3(halfMagnitude, 0, -halfMagnitude);
        case "SW":
            return currentPosition - new Vector3(halfMagnitude, 0, halfMagnitude);
        case "NW":
            return currentPosition - new Vector3(halfMagnitude, 0, -halfMagnitude);
        default:
            return currentPosition; // Return current position if the direction is invalid
    }
}

private void UpdateActionText(string text)
{
    actionText.text = text;
    Debug.Log($"Action text updated to: {actionText.text}");
}
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 5

CubeController

```
using UnityEngine;

public class CubeController : MonoBehaviour
{
   // Start is called before the first frame update
   void Start()
   {
      // Initialize the cube's position at (0,0,0)
      transform.position = new Vector3(0, 0, 0);
      Debug.Log("Cube initialized at position (0, 0, 0)");
   }

   // Method to move the cube to a new position using float coordinates
   public void MoveCube(float x, float z)
   {
      // Set the new position of the cube using the provided x and z coordinates
      transform.position = new Vector3(x, 0, z);
      Debug.Log($"Cube moved to position ({x}, 0, {z})");
   }
}
```

ButtonManager

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

public class ButtonManager : MonoBehaviour
{
   // References to each button object
   public GameObject Cube_ButtonN1;
   public GameObject Cube_ButtonN2;
   public GameObject Cube_ButtonN3;
   public GameObject Cube_ButtonN4;
   public GameObject Cube_ButtonN5;
   public GameObject Cube_ButtonN6;
   public GameObject Cube_ButtonN7;
   public GameObject Cube_ButtonN8;
   public GameObject Cube_ButtonN9;
   public GameObject Cube_ButtonN10;
   public GameObject Cube_ButtonN11;
   public GameObject Cube_ButtonN12;
   public GameObject Cube_ButtonN13;
   public GameObject Cube_ButtonN14;
   public GameObject Cube_ButtonN15;
   public GameObject Cube_ButtonN16;

   // Reference to the CubeController script
   public CubeController cubeController;
   // Reference to the Text UI element
   public TextMeshProUGUI moveText;

   private int? xSelection = null;
   private int? zSelection = null;

   void OnCollisionEnter(Collision collisionInfo)
   {
      // Check each button by name for X-axis selections and log separately
      if (collisionInfo.collider.name == "Cube_ButtonN1")
      {
         xSelection = 0;
         Debug.Log("Collision with: Cube_ButtonN1");
      }
      else if (collisionInfo.collider.name == "Cube_ButtonN2")
      {
         xSelection = 1;
         Debug.Log("Collision with: Cube_ButtonN2");
      }
```

```
else if (collisionInfo.collider.name == "Cube_ButtonN3")
{
   xSelection = 2;
   Debug.Log("Collision with: Cube_ButtonN3");
}
else if (collisionInfo.collider.name == "Cube_ButtonN4")
{
   xSelection = 3;
   Debug.Log("Collision with: Cube_ButtonN4");
}
else if (collisionInfo.collider.name == "Cube_ButtonN5")
{
   xSelection = 4;
   Debug.Log("Collision with: Cube_ButtonN5");
}
else if (collisionInfo.collider.name == "Cube_ButtonN6")
{
   xSelection = 5;
   Debug.Log("Collision with: Cube_ButtonN6");
}
else if (collisionInfo.collider.name == "Cube_ButtonN7")
{
   xSelection = 6;
   Debug.Log("Collision with: Cube_ButtonN7");
}
else if (collisionInfo.collider.name == "Cube_ButtonN8")
{
   xSelection = 7;
   Debug.Log("Collision with: Cube_ButtonN8");
}
// Check each button by name for Z-axis selections and log separately
else if (collisionInfo.collider.name == "Cube_ButtonN9")
{
   zSelection = 0;
   Debug.Log("Collision with: Cube_ButtonN9");
}
else if (collisionInfo.collider.name == "Cube_ButtonN10")
{
   zSelection = 1;
   Debug.Log("Collision with: Cube_ButtonN10");
}
else if (collisionInfo.collider.name == "Cube_ButtonN11")
{
   zSelection = 2;
   Debug.Log("Collision with: Cube_ButtonN11");
}
else if (collisionInfo.collider.name == "Cube_ButtonN12")
```

```
    {
        zSelection = 3;
        Debug.Log("Collision with: Cube_ButtonN12");
    }
    else if (collisionInfo.collider.name == "Cube_ButtonN13")
    {
        zSelection = 4;
        Debug.Log("Collision with: Cube_ButtonN13");
    }
    else if (collisionInfo.collider.name == "Cube_ButtonN14")
    {
        zSelection = 5;
        Debug.Log("Collision with: Cube_ButtonN14");
    }
    else if (collisionInfo.collider.name == "Cube_ButtonN15")
    {
        zSelection = 6;
        Debug.Log("Collision with: Cube_ButtonN15");
    }
    else if (collisionInfo.collider.name == "Cube_ButtonN16")
    {
        zSelection = 7;
        Debug.Log("Collision with: Cube_ButtonN16");
    }

    if (xSelection.HasValue && zSelection.HasValue)
    {
        StartCoroutine(DelayedUpdateCubePosition());
    }
}

IEnumerator DelayedUpdateCubePosition()
{
    // Wait for 3 seconds
    yield return new WaitForSeconds(3);

    if (xSelection.HasValue && zSelection.HasValue)
    {
        // Calculate coordinates
        float xCoordinate = 0.5f * xSelection.Value;
        float zCoordinate = -0.5f * zSelection.Value;

        // Move the cube
        cubeController.MoveCube(xCoordinate, zCoordinate);

        // Update text to "Move(x, z)" based on the selections
        moveText.text = $"Move({xSelection.Value}, {zSelection.Value})";
```

```
            // Reset the selections for the next move
            xSelection = null;
            zSelection = null;
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 6

PinocchioMovement

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI; // Include the TextMeshPro namespace


public class PinocchioMovement : MonoBehaviour
{
   public float walkDuration;
   public float runDuration;
   private Vector3 startPosition = new Vector3(0, 0, 0);
   private Vector3 endPosition = new Vector3(6, 0, 0);
   private bool isMoving = false;
   private float startTime;
   private float speed;

   public LightToggle lightToggle; // Assigned in the inspector
   public AudioSource audioSource; // Assigned an AudioSource component
   public AudioClip redLightSound; // Assign clips for each condition
   public Text scoreText; // Assign this in the inspector


   void Start()
   {
      transform.position = startPosition;
      scoreText.text = "Score: 0/10"; // Initialize the score text at the start
      scoreText.gameObject.SetActive(true); // Show the score text at start
   }

   void Update()
   {
      if (isMoving)
      {
         float timeSinceStarted = Time.time - startTime;
         float fractionOfJourney = timeSinceStarted / speed;
         transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

         // Check if Pinocchio has reached the end position
         if (Vector3.Distance(transform.position, endPosition) < 0.01f)
         {
            isMoving = false;
         }
      }
```

```
    }

    public void StartWalking()
    {
        StartMoving(walkDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() >= 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
        }
        else
        {
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    public void StartRunning()
    {
        StartMoving(runDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() < 5)
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
            else if (lightToggle.GetCurrentCountdown() >= 5 &&
lightToggle.GetCurrentCountdown() < 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            // Modified condition to update the score to 0/10 when the counter is
between 15 and 10
            else if (lightToggle.GetCurrentCountdown() >= 10 &&
lightToggle.GetCurrentCountdown() <= 15)
            {
                UpdateScore(false); // Update score to 0/10
            }
            else
```

```
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
        }
        else
        {
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private void StartMoving(float duration)
    {
        if (!isMoving)
        {
            isMoving = true;
            startTime = Time.time;
            speed = 1.0f / duration;
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }

    private void UpdateScore(bool conditionMet)
    {
        if (conditionMet)
            scoreText.text = "Score: 10/10";
        else
            scoreText.text = "Score: 0/10";
    }
}
```

CollisionHandler

```
using UnityEngine;

public class CollisionHandler : MonoBehaviour
{
   public PinocchioMovement targetScript;  // Script that contains the movement
logic
   public GameObject Button_walk;
   public GameObject Button_run;


   void OnCollisionEnter(Collision collisionInfo)
   {
      if (collisionInfo.collider.name == "Cube_Button1")
      {
         targetScript.StartWalking();
      }
      else if (collisionInfo.collider.name == "Cube_Button3")
      {
         targetScript.StartRunning();
      }
   }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 7

CollisionHandler_204

```
using UnityEngine;

public class CollisionHandler_204 : MonoBehaviour
{
   public PinocchioMovement_204 targetScript1;
   public PinocchioMovement_204_2 targetScript2;
   public PinocchioMovement_204_3 targetScript3; // Script that contains the
movement logic
   public GameObject Button_walk_1;
   public GameObject Button_run_1;
   public GameObject Button_walk_2;
   public GameObject Button_run_2;
   public GameObject Button_walk_3;
   public GameObject Button_run_3;



   void OnCollisionEnter(Collision collisionInfo)
   {
      if (collisionInfo.collider.name == "Cube_Button_walk_1")
      {
         targetScript1.StartWalking_204();
      }
      if (collisionInfo.collider.name == "Cube_Button_run_1")
      {
         targetScript1.StartRunning_204();
      }
      if (collisionInfo.collider.name == "Cube_Button_walk_2")
      {
         targetScript2.StartWalking_204_2();
      }
      if (collisionInfo.collider.name == "Cube_Button_run_2")
      {
         targetScript2.StartRunning_204_2();
      }
      if (collisionInfo.collider.name == "Cube_Button_walk_3")
      {
         targetScript3.StartWalking_204_3();
      }
      if (collisionInfo.collider.name == "Cube_Button_run_3")
      {
         targetScript3.StartRunning_204_3();
      }
```

```
    }
}
```

PinocchioMovement_204_3

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204_3 : MonoBehaviour
{
    public float walkDuration;
    public float runDuration;
    private Vector3 startPosition = new Vector3(23, 0, 0);
    private Vector3 endPosition = new Vector3(29, 0, 0);
    private bool isMoving = false;
    private float startTime;
    private float duration;

    public LightToggle_204_3 lightToggle_204_3; // Assigned in the inspector
    public AudioSource audioSource; // Assigned an AudioSource component
    public AudioClip redLightSound; // Assign clips for each condition

    void Start()
    {
        Debug.Log("PinocchioMovement_204_3 initialized.");
    }

    void Update()
    {
        if (isMoving)
        {
            MovePinocchio();
        }
    }

    public void StartWalking_204_3()
    {
        if (Vector3.Distance(transform.position, startPosition) < 0.5f)
        {
            StartMoving(walkDuration);
            EvaluateConditions(isWalking: true);
        }
    }

    public void StartRunning_204_3()
    {
        if (Vector3.Distance(transform.position, startPosition) < 0.5f)
        {
            StartMoving(runDuration);
            EvaluateConditions(isWalking: false);
        }
```

```
    }

    private void StartMoving(float duration)
    {
        isMoving = true;
        startTime = Time.time;
        this.duration = duration;
    }

    private void MovePinocchio()
    {
        float timeSinceStarted = Time.time - startTime;
        float fractionOfJourney = timeSinceStarted / duration;
        float smoothFraction = Mathf.SmoothStep(0, 1, fractionOfJourney);
        transform.position = Vector3.Lerp(startPosition, endPosition, smoothFraction);

        if (fractionOfJourney >= 1.0f)
        {
            isMoving = false;
            Debug.Log("Movement completed.");
        }
    }

    private void EvaluateConditions(bool isWalking)
    {
        Debug.Log("Evaluating light and score conditions.");
        if (lightToggle_204_3.IsGreenLight())
        {
            int countdown = lightToggle_204_3.GetCurrentCountdown();
            if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
            {
                ScoreManager.Instance.AddScore(10); // Gain 10 points
            }
        }
        else
        {
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }
}
```

PinocchioMovement_204

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204 : MonoBehaviour
{
   public float walkDuration;
   public float runDuration;
   private Vector3 startPosition = new Vector3(0, 0, 0);
   private Vector3 endPosition = new Vector3(6, 0, 0);
   private bool isMoving = false;
   private float startTime;
   private float speed;

   public LightToggle_204 lightToggle_204; // Assigned in the inspector
   public AudioSource audioSource; // Assigned an AudioSource component
   public AudioClip redLightSound; // Assign clips for each condition
   public AudioClip urgentMoveSound; // Assign a sound clip for urgent movement
under green light with countdown < 5

   void Start()
   {
      transform.position = startPosition;
      Debug.Log("Pinocchio start position set.");
   }

   void Update()
   {
      if (!enabled)
      {
         Debug.Log("PinocchioMovement_204 script is disabled.");
         return; // Do not execute if the script is disabled
      }

      if (isMoving)
      {
         float timeSinceStarted = Time.time - startTime;
         float fractionOfJourney = timeSinceStarted / speed;
         transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

         Debug.Log("Pinocchio is moving towards end position.");

         // Check if Pinocchio has reached the end position
         if (Vector3.Distance(transform.position, endPosition) < 0.01f)
         {
            isMoving = false;
```

```
            Debug.Log("Pinocchio reached the end position.");
        }
    }
}

public void StartWalking_204()
{
    if (!enabled)
    {
        Debug.Log("Attempt to walk while script is disabled.");
        return; // Do not execute if the script is disabled
    }

    Debug.Log("Pinocchio starts walking.");
    StartMoving(walkDuration);
    EvaluateConditions(isWalking: true);
}

public void StartRunning_204()
{
    if (!enabled)
    {
        Debug.Log("Attempt to run while script is disabled.");
        return; // Do not execute if the script is disabled
    }

    Debug.Log("Pinocchio starts running.");
    StartMoving(runDuration);
    EvaluateConditions(isWalking: false);
}

private void StartMoving(float duration)
{
    if (!isMoving)
    {
        isMoving = true;
        startTime = Time.time;
        speed = 1.0f / duration;
        Debug.Log("Movement initialized with duration: " + duration.ToString());
    }
}

private void EvaluateConditions(bool isWalking)
{
    Debug.Log("Evaluating light and score conditions.");
    if (lightToggle_204.IsGreenLight())
    {
```

```
        int countdown = lightToggle_204.GetCurrentCountdown();
        if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
        {
            ScoreManager.Instance.AddScore(10); // Gain 10 points
        }

        // Play urgentMoveSound if the countdown is less than 5
        if (countdown < 5)
        {
            audioSource.PlayOneShot(urgentMoveSound);
            Debug.Log("Playing urgent move sound due to low countdown under green
light.");
        }
      }
      else
      {
        StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
      }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
      Debug.Log("Playing sound after delay.");
      yield return new WaitForSeconds(delay);
      audioSource.PlayOneShot(clip);
    }
}
```

PinocchioAutoMove

```
using UnityEngine;

public class PinocchioAutoMove : MonoBehaviour
{
   public float autoMoveSpeed = 100f;  // Speed for automatic movement, adjustable
in the inspector
   public float waitTime = 0f;       // Time to wait at the target position before moving,
adjustable in the inspector
   private Vector3 targetPosition = new Vector3(11, 0, 0);
   private Vector3 checkPosition = new Vector3(6, 0, 0);
   private bool shouldMove = false;
   private bool isWaiting = false;
   private float startTime;
   private float waitStartTime;
   private float journeyLength;


   void Awake()
   {
      Debug.Log("Script initialized on Awake automove");
   }
   void Start()
   {
      journeyLength = Vector3.Distance(checkPosition, targetPosition);

      Debug.Log("Script initialized on Start automove ");
   }

   void Update()
   {
      // Check if Pinocchio is at the specified position and not already moving
      if (Vector3.Distance(transform.position, checkPosition) < 0.01f && !shouldMove
&& !isWaiting)
      {
         shouldMove = true;
         isWaiting = true;
         waitStartTime = Time.time;  // Start the wait timer
      }

      // Handle waiting at the check position
      if (isWaiting && (Time.time - waitStartTime >= waitTime))
      {
         isWaiting = false;
         startTime = Time.time;  // Reset start time for movement after waiting
      }
```

```
        // Proceed with movement after waiting
        if (shouldMove && !isWaiting)
        {
            float distCovered = (Time.time - startTime) * autoMoveSpeed;
            float fractionOfJourney = distCovered / journeyLength;
            transform.position = Vector3.Lerp(checkPosition, targetPosition,
fractionOfJourney);

            // Check if Pinocchio has reached the target position
            if (Vector3.Distance(transform.position, targetPosition) < 0.01f)
            {
                shouldMove = false;  // Stop moving once the target is reached
            }
        }
    }
}
```

PinocchioMovement_204_2

```
using System.Collections;
using UnityEngine;

public class PinocchioMovement_204_2 : MonoBehaviour
{
   public float walkDuration;
   public float runDuration;
   private Vector3 startPosition = new Vector3(11, 0, 0);
   private Vector3 endPosition = new Vector3(17, 0, 0);
   private bool isMoving = false;
   private float startTime;
   private float duration;

   public LightToggle_204_2 lightToggle_204_2; // Assigned in the inspector
   public AudioSource audioSource; // Assigned an AudioSource component
   public AudioClip redLightSound; // Assign clips for each condition



   void Update()
   {
      if (isMoving)
      {
         MovePinocchio();
      }
   }

   public void StartWalking_204_2()
   {
      if (Vector3.Distance(transform.position, startPosition) < 0.5f)
      {
         StartMoving(walkDuration);
         EvaluateConditions(isWalking: true);
      }
   }

   public void StartRunning_204_2()
   {
      if (Vector3.Distance(transform.position, startPosition) < 0.5f)
      {
         StartMoving(runDuration);
         EvaluateConditions(isWalking: false);
      }
   }

   private void StartMoving(float duration)
```

```
    {
        isMoving = true;
        startTime = Time.time;
        this.duration = duration;
    }

    private void MovePinocchio()
    {
        float timeSinceStarted = Time.time - startTime;
        float fractionOfJourney = timeSinceStarted / duration;
        float smoothFraction = Mathf.SmoothStep(0, 1, fractionOfJourney);
        transform.position = Vector3.Lerp(startPosition, endPosition, smoothFraction);

        if (fractionOfJourney >= 1.0f)
        {
            isMoving = false;
            Debug.Log("Movement completed.");
        }
    }

    private void EvaluateConditions(bool isWalking)
    {
        Debug.Log("Evaluating light and score conditions.");
        if (lightToggle_204_2.IsGreenLight())
        {
            int countdown = lightToggle_204_2.GetCurrentCountdown();
            if ((isWalking && countdown >= 10) || (!isWalking && countdown < 10 &&
countdown >= 5))
            {
                ScoreManager.Instance.AddScore(10); // Gain 10 points
            }
        }
        else
        {
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }
}
```

PinocchioAutoMove_2

```
using UnityEngine;

public class PinocchioAutoMove_2 : MonoBehaviour
{
   public float autoMoveSpeed = 1f;  // Speed for automatic movement, adjustable in the inspector
   public float waitTime = 2f;       // Time to wait at the target position before moving, adjustable in the inspector
   private Vector3 targetPosition = new Vector3(23, 0, 0);
   private Vector3 checkPosition = new Vector3(17, 0, 0);
   private bool shouldMove = false;
   private bool isWaiting = false;
   private float startTime;
   private float waitStartTime;
   private float journeyLength;

   void Start()
   {
      journeyLength = Vector3.Distance(checkPosition, targetPosition);
   }

   void Update()
   {
      // Check if Pinocchio is at the specified position and not already moving
      if (Vector3.Distance(transform.position, checkPosition) < 0.01f && !shouldMove && !isWaiting)
      {
         shouldMove = true;
         isWaiting = true;
         waitStartTime = Time.time;  // Start the wait timer
      }

      // Handle waiting at the check position
      if (isWaiting && (Time.time - waitStartTime >= waitTime))
      {
         isWaiting = false;
         startTime = Time.time;  // Reset start time for movement after waiting
      }

      // Proceed with movement after waiting
      if (shouldMove && !isWaiting)
      {
         float distCovered = (Time.time - startTime) * autoMoveSpeed;
         float fractionOfJourney = distCovered / journeyLength;
         transform.position = Vector3.Lerp(checkPosition, targetPosition, fractionOfJourney);
```

```
            // Check if Pinocchio has reached the target position
            if (Vector3.Distance(transform.position, targetPosition) < 0.01f)
            {
                shouldMove = false;  // Stop moving once the target is reached
            }
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 8

IndividualCubeController

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI; // Include the TextMeshPro namespace


public class IndividualCubeController : MonoBehaviour
{
    // References to each individual cube, grouped by button
    public GameObject cube5_1, cube5_2, cube5_3, cube5_4, cube5_5;
    public GameObject cube4_1, cube4_2, cube4_3, cube4_4;
    public GameObject cube3_1, cube3_2, cube3_3;
    public GameObject cube2_1, cube2_2;
    public GameObject cube1_1;

    // Single cubes representing button inputs
    public GameObject Cube_Button1;
    public GameObject Cube_Button2;
    public GameObject Cube_Button3;
    public GameObject Cube_Button4;
    public GameObject Cube_Button5;

    // References to each plane
    public GameObject plane4, plane3, plane2, plane1;
    public GameObject plane2_2, plane2_3, plane2_4, plane2_5; // Additional planes

    // References to TextMeshPro Texts
    public Text Text_5, Text_4, Text_3, Text_2, Text_1;

    // Public color variable for cube color changes
    public Color newColorForCubes = Color.red;

    void Start()
    {
        DeactivateAll();  // Deactivate all cubes and planes at the start of the scene
        DeactivateTexts();
    }

    public void ShowCubes(int buttonNumber)
    {
        StartCoroutine(ActivateCubesAndPlanes(buttonNumber));
    }

    private IEnumerator ActivateCubesAndPlanes(int buttonNumber)
```

```
  {
    DeactivateAll();
    DeactivateTexts();

    if (buttonNumber >= 5)
    {
      Text_5.gameObject.SetActive(true);
      ActivateCubes(cube5_1, cube5_2, cube5_3, cube5_4, cube5_5);
      yield return new WaitForSeconds(3f);
      plane4.SetActive(true);
    }

    if (buttonNumber >= 4)
    {
      Text_4.gameObject.SetActive(true);
      ActivateCubes(cube4_1, cube4_2, cube4_3, cube4_4);
      yield return new WaitForSeconds(3f);
      plane3.SetActive(true);
    }

    if (buttonNumber >= 3)
    {
      Text_3.gameObject.SetActive(true);
      ActivateCubes(cube3_1, cube3_2, cube3_3);
      yield return new WaitForSeconds(3f);
      plane2.SetActive(true);
    }

    if (buttonNumber >= 2)
    {
      Text_2.gameObject.SetActive(true);
      ActivateCubes(cube2_1, cube2_2);
      yield return new WaitForSeconds(3f);
      plane1.SetActive(true);
    }

    if (buttonNumber >= 1)
    {
      Text_1.gameObject.SetActive(true);
      ActivateCubes(cube1_1);
      yield return new WaitForSeconds(3f);
      StartCoroutine(DelayedPlaneDeactivation());
    }

    yield return new WaitForSeconds(3f); // Additional delay before showing new
planes
    StartCoroutine(ShowAdditionalPlanes(buttonNumber));
```

```
}

private IEnumerator DelayedPlaneDeactivation()
{
    yield return new WaitForSeconds(3f);
    plane4.SetActive(false);
    plane3.SetActive(false);
    plane2.SetActive(false);
    plane1.SetActive(false);
}

private IEnumerator ShowAdditionalPlanes(int buttonNumber)
{
    // Change color of cube1_1 before showing plane2_2
    ChangeCubeColor(cube1_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    plane2_2.SetActive(true);
    yield return new WaitForSeconds(1f);

    // Change colors of subsequent cubes and show planes with delays
    ChangeCubeColor(cube2_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube2_2, newColorForCubes);
    yield return new WaitForSeconds(2f);

    if (buttonNumber >= 3)
    {
        plane2_3.SetActive(true);
        yield return new WaitForSeconds(1f);
    }

    ChangeCubeColor(cube3_1, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube3_2, newColorForCubes);
    yield return new WaitForSeconds(1f);

    ChangeCubeColor(cube3_3, newColorForCubes);
    yield return new WaitForSeconds(1f);

    if (buttonNumber >= 4)
    {
        plane2_4.SetActive(true);
        yield return new WaitForSeconds(1f);
```

```
            ChangeCubeColor(cube4_1, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube4_2, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube4_3, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube4_4, newColorForCubes);
        }

        if (buttonNumber == 5)
        {
            plane2_5.SetActive(true);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_1, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_2, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_3, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_4, newColorForCubes);
            yield return new WaitForSeconds(1f);

            ChangeCubeColor(cube5_5, newColorForCubes);
        }
    }

    private void DeactivateAll()
    {
        // Deactivate all cubes and planes
        cube5_1.SetActive(false); cube5_2.SetActive(false); cube5_3.SetActive(false);
cube5_4.SetActive(false); cube5_5.SetActive(false);
        cube4_1.SetActive(false); cube4_2.SetActive(false); cube4_3.SetActive(false);
cube4_4.SetActive(false);
        cube3_1.SetActive(false); cube3_2.SetActive(false); cube3_3.SetActive(false);
        cube2_1.SetActive(false); cube2_2.SetActive(false);
        cube1_1.SetActive(false);
        plane4.SetActive(false); plane3.SetActive(false); plane2.SetActive(false);
plane1.SetActive(false);
        plane2_2.SetActive(false); plane2_3.SetActive(false); plane2_4.SetActive(false);
plane2_5.SetActive(false);
```

```
}

private void DeactivateTexts()
{
   // Deactivate all texts
   Text_5.gameObject.SetActive(false);
   Text_4.gameObject.SetActive(false);
   Text_3.gameObject.SetActive(false);
   Text_2.gameObject.SetActive(false);
   Text_1.gameObject.SetActive(false);
}

private void ActivateCubes(params GameObject[] cubes)
{
   foreach (var cube in cubes)
   {
      cube.SetActive(true);
   }
}

private void ChangeCubeColor(GameObject cube, Color color)
{
   Renderer renderer = cube.GetComponent<Renderer>();
   if (renderer != null)
   {
      renderer.material.color = color;
   }
}

void OnCollisionEnter(Collision collisionInfo)
{
   if (collisionInfo.collider.name == "Cube_Button1")
   {
      ShowCubes(1);
   }
   else if (collisionInfo.collider.name == "Cube_Button2")
   {
      ShowCubes(2);
   }
   else if (collisionInfo.collider.name == "Cube_Button3")
   {
      ShowCubes(3);
   }
   else if (collisionInfo.collider.name == "Cube_Button4")
   {
      ShowCubes(4);
   }
```

```
        else if (collisionInfo.collider.name == "Cube_Button5")
        {
            ShowCubes(5);
        }
    }
}
```

IMMERSIVE VR VRBCS 2

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 1

MoveAround12

```
using UnityEngine;
using System;
using System.Collections;

public class MoveAround12 : MonoBehaviour
{
   public GameObject cube; // Assign your Cube GameObject in the inspector

   private Vector3[] allowedPositions = new Vector3[]
   {
      new Vector3(14, 0.08f, 10), new Vector3(13, 0.08f, 10), new Vector3(12, 0.08f,
10),
      new Vector3(11, 0.08f, 10), new Vector3(10, 0.08f, 10), new Vector3(9, 0.08f,
10),
      new Vector3(8, 0.08f, 10), new Vector3(7, 0.08f, 10), new Vector3(6, 0.08f, 10),
      new Vector3(10, 0.08f, 14), new Vector3(10, 0.08f, 13), new Vector3(10, 0.08f,
12),
      new Vector3(10, 0.08f, 11), new Vector3(10, 0.08f, 9), new Vector3(10, 0.08f, 8),
      new Vector3(10, 0.08f, 7), new Vector3(10, 0.08f, 6)
   };

   // Movement state
   private bool isMoving = false;

   // Forward+3 trigger state (existing)
   private bool watchForward3 = false;

   // Backward+6 to B3 trigger state (existing)
   private bool watchBackward6ToB3 = false;

   // Events:
   public event Action ArrivedForward3Now;   // immediate on F3
   public event Action ReachedForward3;      // 2s after F3 (text switch)
   public event Action ArrivedB3Now;         // immediate on B3

   // NEW: per-step event (stepsCompleted from 1..magnitude,
totalSteps==magnitude)
   public event Action<int, int> StepAdvanced;

   // Grid references
```

```
    private static readonly Vector3 Center = new Vector3(10f, 0.08f, 10f);
    private static readonly Vector3 F3Pos = Center + Vector3.right * 3f; // (13, 0.08, 10)
    private static readonly Vector3 B3Pos = Center + Vector3.left * 3f;  // (7,  0.08, 10)

    void Start()
    {
        // Set the cube's starting position and rotation
        cube.transform.position = Center;
        cube.transform.rotation = Quaternion.Euler(0f, 270f, 0f);
        Debug.Log("Cube Start Position: " + cube.transform.position + " | Start Rotation:
" + cube.transform.rotation.eulerAngles);
    }

    // Called by MoveCubeAround when user does Forward + 3 + Run
    public void ArmForward3Trigger()
    {
        watchForward3 = true;
    }

    // Called by MoveCubeAround after Text2 appears to allow Backward + 6 + Run
    public void ArmBackward6ToB3Trigger()
    {
        watchBackward6ToB3 = true;
    }

    public void MoveCube(string direction, int magnitude)
    {
        if (isMoving)
        {
            Debug.Log("Move request ignored: already moving.");
            return;
        }

        // Determine 1-unit step direction for pre-check
        Vector3 step = Vector3.zero;
        switch (direction)
        {
            case "forward": step = Vector3.right; break;  // +X
            case "backward": step = Vector3.left; break;  // -X
            case "right": step = Vector3.back; break;     // -Z
            case "left": step = Vector3.forward; break;   // +Z
        }

        // All-or-nothing pre-check — if any step is invalid, don't move at all
        if (!CanCompleteMove(cube.transform.position, step, magnitude))
        {
```

```
            Debug.Log($"Move aborted: cannot complete full path {direction}
x{magnitude} from {cube.transform.position}.");
            return;
        }

        StartCoroutine(MoveCubeCoroutine(direction, magnitude));
    }

    private IEnumerator MoveCubeCoroutine(string direction, int magnitude)
    {
        isMoving = true;

        // Smoothly rotate to face the intended direction during the existing 3s pre-
move wait
        Quaternion targetRot = GetTargetRotation(direction);
        StartCoroutine(RotateTo(targetRot, 0.5f)); // smooth turn; completes within the
3s wait

        // Wait 3 seconds before starting any movement
        yield return new WaitForSecondsRealtime(3f);

        // Determine 1-unit step direction
        Vector3 step = Vector3.zero;
        switch (direction)
        {
            case "forward": step = Vector3.right; break; // +X
            case "backward": step = Vector3.left; break; // -X
            case "right": step = Vector3.back; break; // -Z
            case "left": step = Vector3.forward; break; // +Z
        }

        for (int i = 0; i < magnitude; i++)
        {
            Vector3 proposed = step;
            Debug.Log($"Attempting to move: {direction} step {i + 1}/{magnitude}.
Proposed step: {proposed}");

            if (CanMove(proposed))
            {
                cube.transform.position += proposed;
                Debug.Log($"Moved: {direction} to new position
{cube.transform.position}");

                // Per-step notification
                StepAdvanced?.Invoke(i + 1, magnitude);

                // -------- F3 trigger (only when actually at F3) --------
```

```
            if (watchForward3 && i == magnitude - 1)
            {
                watchForward3 = false; // one-shot (disarm regardless)
                if (Approximately(cube.transform.position, F3Pos))
                {
                    ArrivedForward3Now?.Invoke();            // immediate (sound)
                    StartCoroutine(FireReachedF3AfterDelay()); // +2s (text switch)
                }
            }

            // -------- B3 trigger --------
            if (watchBackward6ToB3 && direction == "backward" && magnitude == 6
&& i == magnitude - 1)
            {
                if (Approximately(cube.transform.position, B3Pos))
                {
                    watchBackward6ToB3 = false; // one-shot
                    ArrivedB3Now?.Invoke();     // play B3 sound in UI script
                }
                else
                {
                    Debug.Log("Backward+6 completed but final position is not B3; no B3
event fired.");
                }
            }

            // stay 1 second at each tile
            yield return new WaitForSecondsRealtime(1f);
        }
        else
        {
            Debug.Log($"Move blocked during execution: {direction} at position
{cube.transform.position + proposed}");
            break;
        }
    }

    isMoving = false;
}

private IEnumerator FireReachedF3AfterDelay()
{
    yield return new WaitForSecondsRealtime(2f);
    ReachedForward3?.Invoke();
}

private bool CanMove(Vector3 proposedMove)
```

```
    {
        Vector3 newPosition = cube.transform.position + proposedMove;
        foreach (Vector3 pos in allowedPositions)
        {
            if (Vector3.Distance(newPosition, pos) < 0.1f) // Tolerance for floating-point
precision
            {
                return true;
            }
        }
        return false; // Block move if it does not exactly match any allowed position
    }

    // helper — check the entire intended path before moving (all-or-nothing)
    private bool CanCompleteMove(Vector3 startPos, Vector3 step, int magnitude)
    {
        if (magnitude <= 0) return false;
        if (step == Vector3.zero) return false;

        Vector3 probe = startPos;
        for (int i = 0; i < magnitude; i++)
        {
            Vector3 next = probe + step;
            bool allowed = false;
            foreach (Vector3 pos in allowedPositions)
            {
                if (Vector3.Distance(next, pos) < 0.1f)
                {
                    allowed = true;
                    break;
                }
            }
            if (!allowed) return false;
            probe = next;
        }
        return true;
    }

    private static bool Approximately(Vector3 a, Vector3 b)
    {
        return Vector3.Distance(a, b) < 0.11f; // a hair above 0.1 to match
allowedPositions tolerance
    }

    // --- rotation helpers (NEW) ---

    private Quaternion GetTargetRotation(string direction)
```

```
    {
        // forward -> (0,90,0) | right -> (0,180,0) | backward -> (0,270,0) | left -> (0,0,0)
        switch (direction)
        {
            case "forward": return Quaternion.Euler(0f, 90f, 0f);
            case "right": return Quaternion.Euler(0f, 180f, 0f);
            case "backward": return Quaternion.Euler(0f, 270f, 0f);
            case "left": return Quaternion.Euler(0f, 0f, 0f);
            default: return cube.transform.rotation;
        }
    }

    private IEnumerator RotateTo(Quaternion target, float duration)
    {
        Quaternion initial = cube.transform.rotation;
        float t = 0f;
        while (t < duration)
        {
            t += Time.unscaledDeltaTime; // match WaitForSecondsRealtime behavior
            float k = Mathf.Clamp01(t / duration);
            cube.transform.rotation = Quaternion.Slerp(initial, target, k);
            yield return null;
        }
        cube.transform.rotation = target;
    }
}
```

MoveCubeAround12

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using System;           // for Action/event
using System.Collections;

// Include the UI namespace to use Text

public class MoveCubeAround12 : MonoBehaviour
{
    public GameObject Cube_Button1;  // Button to move the cube forward
    public GameObject Cube_Button2;  // Button to move the cube right
    public GameObject Cube_Button3;  // Button to move the cube backward
    public GameObject Cube_Button4;  // Button to move the cube left
    public GameObject Cube_Button5;  // Run button to confirm and move the cube

    // Buttons for specifying movement magnitude
    public GameObject Cube_ButtonN1;
    public GameObject Cube_ButtonN2;
    public GameObject Cube_ButtonN3;
```

```
    public GameObject Cube_ButtonN4;
    public GameObject Cube_ButtonN5;
    public GameObject Cube_ButtonN6;
    public GameObject Cube_ButtonN7;
    public GameObject Cube_ButtonN8;

    // Movement script
    public MoveAround12 targetScript;  // Script that contains the movement logic

    public TextMeshProUGUI actionText;  // Reference to the TextMeshPro UI
component

    // Texts (assign in Inspector)
    public TextMeshProUGUI text1; // Appears at start of scene
    public TextMeshProUGUI text2; // Appears 2s after arriving at F3

    // NEW: Per-move countdown text (assign in Inspector)
    public TextMeshProUGUI stepCountdownText;

    // Audio
    public AudioSource f3Audio;   // plays when arriving at F3 (optional)
    public AudioSource b3Audio;   // plays when arriving at B3 (optional)

    private bool readyToMoveForward = false;
    private bool readyToMoveBackward = false;
    private bool readyToMoveRight = false;
    private bool readyToMoveLeft = false;
    private int moveMagnitude = 0;  // To store the movement magnitude

    // Gate that allows Backward+6->B3 only after Text2 has appeared
    private bool canAcceptB3 = false;

    void Start()
    {
        if (text1 != null) text1.gameObject.SetActive(true);
        if (text2 != null) text2.gameObject.SetActive(false);
        if (stepCountdownText != null) stepCountdownText.gameObject.SetActive(false);
    }

    void OnEnable()
    {
        if (targetScript != null)
        {
            // idempotent subscribe
            targetScript.ArrivedForward3Now -= HandleArrivedF3Now;
            targetScript.ArrivedForward3Now += HandleArrivedF3Now;
```

```
            targetScript.ReachedForward3 -= HandleReachedF3;
            targetScript.ReachedForward3 += HandleReachedF3;

            targetScript.ArrivedB3Now -= HandleArrivedB3Now;
            targetScript.ArrivedB3Now += HandleArrivedB3Now;

            // NEW: per-step countdown subscription
            targetScript.StepAdvanced -= HandleStepAdvanced;
            targetScript.StepAdvanced += HandleStepAdvanced;
        }
    }

    void OnDisable()
    {
        if (targetScript != null)
        {
            targetScript.ArrivedForward3Now -= HandleArrivedF3Now;
            targetScript.ReachedForward3 -= HandleReachedF3;
            targetScript.ArrivedB3Now -= HandleArrivedB3Now;
            targetScript.StepAdvanced -= HandleStepAdvanced;
        }
    }

    void OnCollisionEnter(Collision collisionInfo)
    {
        // Handle directional buttons
        if (collisionInfo.collider.name == "Cube_Button1")
            readyToMoveForward = true;
        else if (collisionInfo.collider.name == "Cube_Button2")
            readyToMoveRight = true;
        else if (collisionInfo.collider.name == "Cube_Button3")
            readyToMoveBackward = true;
        else if (collisionInfo.collider.name == "Cube_Button4")
            readyToMoveLeft = true;

        // Handle magnitude buttons
        if (collisionInfo.collider.name.StartsWith("Cube_ButtonN"))
        {
            moveMagnitude = int.Parse(collisionInfo.collider.name.Substring(12));  //
Extracting the number from the name
        }

        // Run button
        if (collisionInfo.collider.name == "Cube_Button5" && moveMagnitude > 0)
        {
            // ---- Phase 1: Forward + 3 + Run -> head to F3 ----
            if (readyToMoveForward && moveMagnitude == 3 && targetScript != null)
```

```
        {
            targetScript.ArmForward3Trigger();

            targetScript.MoveCube("forward", moveMagnitude);
            if (actionText != null) actionText.text = $"MoveForward({moveMagnitude})";

            ResetReadyStates();
            return;
        }

        // ---- Phase 2: (after Text2 shown) Backward + 6 + Run -> go to B3 ----
        if (canAcceptB3 && readyToMoveBackward && moveMagnitude == 6 &&
targetScript != null)
        {
            targetScript.ArmBackward6ToB3Trigger();

            targetScript.MoveCube("backward", moveMagnitude);
            if (actionText != null) actionText.text =
$"MoveBackward({moveMagnitude})";

            // One-shot gate; remove if you want to allow retries without re-showing
Text2
            canAcceptB3 = false;

            ResetReadyStates();
            return;
        }

        // ---- General fallback: allow any direction + magnitude (now includes
FORWARD) ----
        if (readyToMoveForward)
        {
            targetScript.MoveCube("forward", moveMagnitude);
            if (actionText != null) actionText.text = $"MoveForward({moveMagnitude})";
        }
        else if (readyToMoveBackward)
        {
            targetScript.MoveCube("backward", moveMagnitude);
            if (actionText != null) actionText.text =
$"MoveBackward({moveMagnitude})";
        }
        else if (readyToMoveRight)
        {
            targetScript.MoveCube("right", moveMagnitude);
            if (actionText != null) actionText.text = $"MoveRight({moveMagnitude})";
        }
        else if (readyToMoveLeft)
```

```
        {
            targetScript.MoveCube("left", moveMagnitude);
            if (actionText != null) actionText.text = $"MoveLeft({moveMagnitude})";
        }

        ResetReadyStates();
    }
}

// Immediate on-arrival at F3 (optional sound)
private void HandleArrivedF3Now()
{
    if (f3Audio != null) f3Audio.Play();
    // leave subscription as OnEnable/OnDisable handle lifecycle
}

// Delayed (2s) after arriving at F3: show Text2 and hide Text1, then enable the B3
phase
private void HandleReachedF3()
{
    if (text1 != null) text1.gameObject.SetActive(false);
    if (text2 != null) text2.gameObject.SetActive(true);
    canAcceptB3 = true;
}

// Immediate on-arrival at B3: play B3 sound
private void HandleArrivedB3Now()
{
    if (b3Audio != null) b3Audio.Play();
}

// NEW: update countdown text every successful step
private void HandleStepAdvanced(int stepsCompleted, int totalSteps)
{
    if (stepCountdownText == null) return;

    int remaining = Mathf.Max(0, totalSteps - stepsCompleted); // e.g., 3->2->1->0
    stepCountdownText.gameObject.SetActive(true);
    stepCountdownText.text = remaining.ToString();
}

private void ResetReadyStates()
{
    readyToMoveForward = false;
    readyToMoveBackward = false;
    readyToMoveRight = false;
    readyToMoveLeft = false;
```

```
        moveMagnitude = 0;  // Reset magnitude
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 2

VRButtonCollisionDispatcher14

```
using UnityEngine;

/// Assign each object ONCE in the Inspector:
/// - cube (CubePathMover on your cube)
/// - start_1, start_2, right, left, run, N1..N8 (button GameObjects)
/// Logs every hit and forwards calls to the cube.
public class VRButtonCollisionDispatcher14 : MonoBehaviour
{
   [Header("Target (assign once)")]
   public CubePathMover14 cube;   // the cube controller script (on your cube)

   [Header("Buttons (assign each once)")]
   public GameObject start_1;   // start phase 1
   public GameObject start_2;   // start phase 2 (Text_4 on, delayed move to l2)
   public GameObject right;     // "right"
   public GameObject left;      // "left"
   public GameObject run;       // "run"
   public GameObject N1;        // magnitude 1
   public GameObject N2;        // magnitude 2
   public GameObject N3;        // magnitude 3
   public GameObject N4;        // magnitude 4
   public GameObject N5;        // magnitude 5
   public GameObject N6;        // magnitude 6
   public GameObject N7;        // magnitude 7
   public GameObject N8;        // magnitude 8

   [Header("Logging")]
   public bool logStayEvents = false;

   private void Awake()
   {
      Debug.Log($"[VRDispatcher] Awake on '{name}' (tag:{tag},
id:{GetInstanceID()})");
   }

   private void Start()
   {
      if (cube == null) Debug.LogWarning("[VRDispatcher] 'cube' is NOT assigned in
the Inspector.");
   }

   // -------- Trigger path --------
   private void OnTriggerEnter(Collider other)
   {
```

```
        Debug.Log($"[VRDispatcher][TriggerEnter] '{name}' hit '{other.name}'
(tag:{other.tag}, id:{other.GetInstanceID()})");
        Route(other.gameObject, "TriggerEnter");
    }

    private void OnTriggerStay(Collider other)
    {
        if (logStayEvents) Debug.Log($"[VRDispatcher][TriggerStay] '{name}' with
'{other.name}'");
    }

    private void OnTriggerExit(Collider other)
    {
        Debug.Log($"[VRDispatcher][TriggerExit] '{name}' left '{other.name}'");
    }

    // -------- Collision path --------
    private void OnCollisionEnter(Collision collision)
    {
        var go = collision.collider ? collision.collider.gameObject : null;
        string otherName = go ? go.name : "NULL";
        Debug.Log($"[VRDispatcher][CollisionEnter] '{name}' hit '{otherName}',
contacts:{collision.contactCount}, relVel:{collision.relativeVelocity}");
        Route(go, "CollisionEnter");
    }

    private void OnCollisionStay(Collision collision)
    {
        if (logStayEvents && collision.collider)
            Debug.Log($"[VRDispatcher][CollisionStay] '{name}' with
'{collision.collider.name}'");
    }

    private void OnCollisionExit(Collision collision)
    {
        if (collision.collider)
            Debug.Log($"[VRDispatcher][CollisionExit] '{name}' left
'{collision.collider.name}'");
    }

    // -------- Core routing (exact object matches only) --------
    private void Route(GameObject hit, string channel)
    {
        if (hit == null) { Debug.LogWarning("[VRDispatcher] Route called with NULL
GameObject"); return; }
        if (cube == null) { Debug.LogWarning("[VRDispatcher] 'cube' not assigned;
ignoring hit."); return; }
```

```
        // Phase starts
        if (hit == start_1) { LogPress("START_1", channel, hit);
cube.TriggerStart1Sequence(); cube.LogState("after START_1"); return; }
        if (hit == start_2) { LogPress("START_2", channel, hit);
cube.TriggerStart2Sequence(); cube.LogState("after START_2"); return; }

        // Directions
        if (hit == right) { LogPress("RIGHT", channel, hit); cube.ReportRightPressed();
cube.LogState("after RIGHT"); return; }
        if (hit == left)  { LogPress("LEFT", channel, hit); cube.ReportLeftPressed();
cube.LogState("after LEFT");  return; }

        // Run
        if (hit == run) { LogPress("RUN", channel, hit); cube.ReportRunPressed();
cube.LogState("after RUN"); return; }

        // Magnitudes
        if (hit == N1) { LogPress("N1", channel, hit); cube.ReportMagnitudePressed(1);
cube.LogState("after N1"); return; }
        if (hit == N2) { LogPress("N2", channel, hit); cube.ReportMagnitudePressed(2);
cube.LogState("after N2"); return; }
        if (hit == N3) { LogPress("N3", channel, hit); cube.ReportMagnitudePressed(3);
cube.LogState("after N3"); return; }
        if (hit == N4) { LogPress("N4", channel, hit); cube.ReportMagnitudePressed(4);
cube.LogState("after N4"); return; }
        if (hit == N5) { LogPress("N5", channel, hit); cube.ReportMagnitudePressed(5);
cube.LogState("after N5"); return; }
        if (hit == N6) { LogPress("N6", channel, hit); cube.ReportMagnitudePressed(6);
cube.LogState("after N6"); return; }
        if (hit == N7) { LogPress("N7", channel, hit); cube.ReportMagnitudePressed(7);
cube.LogState("after N7"); return; }
        if (hit == N8) { LogPress("N8", channel, hit); cube.ReportMagnitudePressed(8);
cube.LogState("after N8"); return; }

        // Unmapped object
        Debug.Log($"[VRDispatcher] {channel}: Hit '{hit.name}' (no mapped action).");
    }

    private void LogPress(string label, string channel, GameObject hit)
    {
        Debug.Log($"[VRDispatcher][PRESSED] {label} at t={Time.time:F2}s via {channel}
(object='{hit.name}', id:{hit.GetInstanceID()})");
    }
}
```

CubePathMover14

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

public class CubePathMover14 : MonoBehaviour
{
   // ----------------- Positions -----------------
   [Header("Positions (World Space)")]
   public Vector3 pos0 = new Vector3(10f, 0.08f, 10f); // start
   public Vector3 r1 = new Vector3(10f, 0.08f, 9f);
   public Vector3 r2 = new Vector3(10f, 0.08f, 8f);
   public Vector3 r3 = new Vector3(10f, 0.08f, 7f);
   public Vector3 l1 = new Vector3(10f, 0.08f, 11f);
   public Vector3 l2 = new Vector3(10f, 0.08f, 12f);

   // ----------------- Phase 1 Timing -----------------
   [Header("Phase 1 Timing (seconds)")]
   [Tooltip("Wait after start_1 before the cube begins moving r1→r2→r3.")]
   public float startDelay = 3f;
   [Tooltip("Wait time the cube stays on r1 and r2.")]
   public float dwellPerStop = 1f;
   [Tooltip("Lerp time to move between points (Phase 1).")]
   public float moveDuration = 0.5f;
   [Tooltip("After arriving at r3, wait this long before showing Text_2 and hiding
Text_1.")]
   public float appearText2Delay = 2f;
   [Tooltip("How long the user has to enter RIGHT+3+RUN after Text_2 appears.")]
   public float comboWindowSeconds_P1 = 8f;

   // ----------------- Phase 2 Timing -----------------
   [Header("Phase 2 Timing (seconds)")]
   [Tooltip("Delay AFTER pressing start_2 before starting Phase 2 movement.")]
   public float start2Delay = 5f;
   [Tooltip("Lerp time to move between points (Phase 2).")]
   public float moveDuration_P2 = 0.5f;
   [Tooltip("Wait time the cube stays on intermediate Phase 2 stops (legacy).")]
   public float dwellPerStop_P2 = 1f; // kept for compatibility (phase-2 uses per-point
dwells below)
   [Tooltip("After arriving at l2, wait this long before showing Text_4.")]
   public float appearText4Delay = 2f;
   [Tooltip("How long the user has to enter LEFT+5+RUN after Text_4 appears.")]
   public float comboWindowSeconds_P2 = 8f;

   // ----------------- Phase 2 Per-Point Dwells -----------------
   [Header("Phase 2 Per-Point Dwells (seconds)")]
   [Tooltip("Stay time at r2 during Phase 2.")]
```

```
    public float dwellAt_r2_P2 = 0.5f;
    [Tooltip("Stay time at r1 during Phase 2.")]
    public float dwellAt_r1_P2 = 0.5f;
    [Tooltip("Stay time at pos0 during Phase 2.")]
    public float dwellAt_pos0_P2 = 0.5f;
    [Tooltip("Stay time at l1 during Phase 2.")]
    public float dwellAt_l1_P2 = 0.5f;

    // ----------------- Rotation -----------------
    [Header("Rotation (smooth pre-rotation before each phase)")]
    [Tooltip("Target rotation (Euler) before Phase 1 movement starts.")]
    public Vector3 rotTarget_P1_Euler = new Vector3(0f, 180f, 0f);
    [Tooltip("Seconds to rotate to Phase 1 target.")]
    public float rotDuration_P1 = 0.5f;

    [Tooltip("Target rotation (Euler) before Phase 2 movement starts.")]
    public Vector3 rotTarget_P2_Euler = new Vector3(0f, 0f, 0f);
    [Tooltip("Seconds to rotate to Phase 2 target.")]
    public float rotDuration_P2 = 0.5f;

    // ----------------- UI -----------------
    [Header("UI (assign in Inspector)")]
    public TextMeshProUGUI Text_1;      // show at start
    public TextMeshProUGUI Text_2;      // appears after arrive r3 + appearText2Delay
    public TextMeshProUGUI Text_3;      // stays visible during P2 until arrive at l2
    public TextMeshProUGUI Text_4;      // appears ONLY after arriving at l2 (Phase 2)
    public TextMeshProUGUI Score_Text;  // shows cumulative score (0/20, 10/20,
20/20)

    // ----------------- Audio -----------------
    [Header("Audio (assign in Inspector)")]
    public AudioSource successSfxPhase1;  // plays on correct RIGHT+3+RUN
    public AudioSource successSfxPhase2;  // plays on correct LEFT+5+RUN

    // ----------------- Behavior -----------------
    [Header("Behavior")]
    public bool smoothStep = true;         // easing for movement & rotation curves
    public bool blockWhileRunning = true;   // ignore new Phase 1 triggers until
sequence is done

    [Header("Debugging")]
    public bool verboseDebug = true;       // toggle detailed state logs

    // ----------------- Internal State -----------------
    private Coroutine runRoutine;     // phase 1 move sequence
    private Coroutine comboRoutine;   // active combo timeout
    private int score = 0;            // 0, 10, 20
```

```
    private enum ComboStage { None, ExpectDir, ExpectMag, ExpectRun }
    private enum ComboDirection { Right, Left }

    private bool inComboWindow = false;
    private bool comboResolved = false;
    private ComboStage comboStage = ComboStage.None;
    private ComboDirection expectedDir;
    private int expectedMagnitude = 0;
    private float currentComboWindowSeconds = 0f;
    private bool phase2Ready = false; // true after P1 resolves & only if user pressed
any P1 input

    private bool anyInputPhase1 = false; // did user press anything during P1 window?
    private int activeComboPhase = 0;    // 0 none, 1 P1, 2 P2

    // Ensure Text_3 and Text_4 are OFF when script enables / scene starts
    private void OnEnable()
    {
        SafeOff(Text_3);
        SafeOff(Text_4);
    }

    private void Start()
    {
        // Scene start pose
        transform.position = pos0;
        transform.rotation = Quaternion.Euler(0f, 270f, 0f);

        Debug.Log($"[CubePathMover] Start: pos={pos0},
rot={transform.rotation.eulerAngles}");

        // Initial UI state
        SafeOn(Text_1);
        SafeOff(Text_2);
        SafeOff(Text_3);
        SafeOff(Text_4);
        SafeOff(Score_Text);
    }

    // -------- Public state logger (called by dispatcher too) --------
    public void LogState(string context = "")
    {
        if (!verboseDebug) return;
        string state = $"phase={activeComboPhase}, stage={comboStage},
inWindow={inComboWindow}, " +
                       $"expectedDir={expectedDir}, expectedMag={expectedMagnitude}, " +
```

```
                $"score={score}/20, phase2Ready={phase2Ready}";
        Debug.Log($"[CubePathMover][STATE] {context} | {state}");
    }

    // ----------------- Phase 1 trigger (from controller: start_1) -----------------
    public void TriggerStart1Sequence()
    {
        Debug.Log("[CubePathMover] TriggerStart1Sequence()");
        if (runRoutine != null && blockWhileRunning)
        {
            Debug.Log("[CubePathMover] Start1 trigger ignored: sequence already
running.");
            return;
        }
        if (runRoutine != null) StopCoroutine(runRoutine);

        runRoutine = StartCoroutine(Start1Sequence());
    }

    private IEnumerator Start1Sequence()
    {
        Debug.Log($"[CubePathMover] Phase 1: wait {startDelay}s.");
        yield return new WaitForSeconds(startDelay);

        // ---- CRITICAL: Smooth pre-rotation at pos0 BEFORE ANY MOVEMENT ----
        Quaternion targetRotP1 = Quaternion.Euler(rotTarget_P1_Euler); // default
(0,180,0)
        Debug.Log($"[CubePathMover] Phase 1 pre-rotate FROM
{transform.rotation.eulerAngles} TO {rotTarget_P1_Euler} in {rotDuration_P1}s.");
        yield return RotateTo(targetRotP1, rotDuration_P1);
        // hard-set exact final orientation
        transform.rotation = targetRotP1;
        Debug.Log($"[CubePathMover] Phase 1 pre-rotate complete. Now at
{transform.rotation.eulerAngles}");

        // ---- Movement sequence r1 → r2 → r3 ----
        Debug.Log("[CubePathMover] Moving to r1...");
        yield return MoveTo(r1, moveDuration);
        Debug.Log($"[CubePathMover] Arrived r1. Dwell {dwellPerStop}s.");
        yield return new WaitForSeconds(dwellPerStop);

        Debug.Log("[CubePathMover] Moving to r2...");
        yield return MoveTo(r2, moveDuration);
        Debug.Log($"[CubePathMover] Arrived r2. Dwell {dwellPerStop}s.");
        yield return new WaitForSeconds(dwellPerStop);

        Debug.Log("[CubePathMover] Moving to r3...");
```

```
        yield return MoveTo(r3, moveDuration);
        Debug.Log("[CubePathMover] Arrived r3.");

        Debug.Log($"[CubePathMover] Wait {appearText2Delay}s then show Text_2.");
        yield return new WaitForSeconds(appearText2Delay);

        SafeOff(Text_1);
        SafeOn(Text_2);
        Debug.Log("[CubePathMover] Text_2 shown. Opening P1 combo
RIGHT→N3→RUN.");
        OpenComboWindow(ComboDirection.Right, 3, comboWindowSeconds_P1,
phase: 1);

        runRoutine = null;
    }

    // ----------------- Movement & Rotation helpers -----------------
    private IEnumerator MoveTo(Vector3 target, float duration)
    {
        Vector3 start = transform.position;

        if (duration <= 0f)
        {
            Debug.LogWarning($"[CubePathMover] duration <= 0 → teleport to {target}");
            transform.position = target;
            yield break;
        }

        float t = 0f;
        Debug.Log($"[CubePathMover] Lerp {start} → {target} over {duration}s
(smoothStep={smoothStep})");
        while (t < duration)
        {
            t += Time.deltaTime;
            float u = Mathf.Clamp01(t / duration);
            if (smoothStep) u = u * u * (3f - 2f * u); // smoothstep easing
            transform.position = Vector3.Lerp(start, target, u);
            yield return null;
        }
        transform.position = target;
        Debug.Log($"[CubePathMover] Reached {target}");
    }

    // Robust smooth rotation that can't "miss"
    private IEnumerator RotateTo(Quaternion target, float duration)
    {
        Quaternion start = transform.rotation;
```

```
        if (duration <= 0f)
        {
            Debug.LogWarning($"[CubePathMover] rot duration <= 0 → snap rotate to
{target.eulerAngles}");
            transform.rotation = target;
            yield break;
        }

        // Compute angular speed so we arrive exactly in 'duration'
        float totalAngle = Quaternion.Angle(start, target);
        float speed = (totalAngle <= 0.01f) ? 0f : totalAngle / duration;

        float t = 0f;
        while (t < duration)
        {
            t += Time.deltaTime;
            // Use RotateTowards for reliability; apply optional easing to 'step' length
            float step = speed * Time.deltaTime;
            if (smoothStep)
            {
                float u = Mathf.Clamp01(t / duration);
                // increase step near the middle, smaller at the ends (approximate ease)
                step *= (u * u * (3f - 2f * u)) * 1.5f + 0.1f;
            }
            transform.rotation = Quaternion.RotateTowards(transform.rotation, target,
step);
            yield return null;
        }
        // ensure exact final target (prevents tiny drift)
        transform.rotation = target;
    }

    // ----------------- Combo System (shared) -----------------
    private void OpenComboWindow(ComboDirection dir, int magnitude, float
windowSeconds, int phase)
    {
        if (comboRoutine != null) StopCoroutine(comboRoutine);

        expectedDir = dir;
        expectedMagnitude = magnitude;
        comboStage = ComboStage.ExpectDir;
        comboResolved = false;
        inComboWindow = true;
        currentComboWindowSeconds = windowSeconds;
        activeComboPhase = phase;
        if (phase == 1) anyInputPhase1 = false;
```

```
        Debug.Log($"[CubePathMover] COMBO OPEN {windowSeconds}s (Phase
{phase}) → expect {dir} → N{magnitude} → RUN.");
        LogState("on open");
        comboRoutine = StartCoroutine(ComboTimeout());
    }

    private IEnumerator ComboTimeout()
    {
        yield return new WaitForSeconds(currentComboWindowSeconds);
        if (!comboResolved)
        {
            Debug.Log("[CubePathMover] COMBO TIMEOUT → FAIL");
            ResolveCombo(false, reason: "timeout");
        }
        comboRoutine = null;
    }

    private void ResolveCombo(bool success, string reason = "")
    {
        if (comboResolved) return;
        comboResolved = true;
        inComboWindow = false;
        comboStage = ComboStage.None;

        if (success)
        {
            score += 10;
            SafeOn(Score_Text);
            if (Score_Text != null) Score_Text.text = $"Score: {score}/20";
            if (activeComboPhase == 1) { if (successSfxPhase1 != null)
successSfxPhase1.Play(); }
            else { if (successSfxPhase2 != null) successSfxPhase2.Play(); }
            Debug.Log($"[CubePathMover] COMBO SUCCESS (+10). New score {score}/20.
{(activeComboPhase == 1 ? "P1" : "P2")} {reason}");
        }
        else
        {
            SafeOn(Score_Text);
            if (Score_Text != null) Score_Text.text = $"Score: {score}/20";
            Debug.Log($"[CubePathMover] COMBO FAIL ({reason}). Score remains
{score}/20 (no sound).");
        }

        // P1 → Text_3 only if user pressed any input in P1
        if (activeComboPhase == 1)
        {
```

```
            if (anyInputPhase1)
            {
                SafeOff(Text_2);
                SafeOn(Text_3);
                phase2Ready = true;
                Debug.Log("[CubePathMover] P1 resolved with input → Text_2 off, Text_3
on. Phase 2 ready.");
            }
            else
            {
                phase2Ready = false;
                Debug.Log("[CubePathMover] P1 resolved by timeout/no input → Text_2
stays, Text_3 stays off.");
            }
        }

        LogState("after resolve");
    }

    // ----------------- Controller inputs -----------------
    public void ReportRightPressed()
    {
        Debug.Log("[CubePathMover][INPUT] RIGHT");
        if (!inComboWindow) { Debug.Log("[CubePathMover] Ignored RIGHT: not in
combo window."); LogState("RIGHT ignored"); return; }
        if (activeComboPhase == 1) anyInputPhase1 = true;

        if (comboStage != ComboStage.ExpectDir || expectedDir !=
ComboDirection.Right)
        {
            Debug.Log($"[CubePathMover] WRONG DIRECTION at RIGHT (expected
{expectedDir}) → FAIL");
            ResolveCombo(false, reason: "wrong direction at first step");
            return;
        }
        comboStage = ComboStage.ExpectMag;
        Debug.Log("[CubePathMover] RIGHT accepted → expect N" +
expectedMagnitude);
        LogState("after RIGHT");
    }

    public void ReportLeftPressed()
    {
        Debug.Log("[CubePathMover][INPUT] LEFT");
        if (!inComboWindow) { Debug.Log("[CubePathMover] Ignored LEFT: not in
combo window."); LogState("LEFT ignored"); return; }
```

```
        if (comboStage != ComboStage.ExpectDir || expectedDir !=
ComboDirection.Left)
        {
            Debug.Log($"[CubePathMover] WRONG DIRECTION at LEFT (expected
{expectedDir}) → FAIL");
            ResolveCombo(false, reason: "wrong direction at first step");
            return;
        }
        comboStage = ComboStage.ExpectMag;
        Debug.Log("[CubePathMover] LEFT accepted → expect N" +
expectedMagnitude);
        LogState("after LEFT");
    }

    public void ReportMagnitudePressed(int n)
    {
        Debug.Log($"[CubePathMover][INPUT] N{n}");
        if (!inComboWindow) { Debug.Log("[CubePathMover] Ignored N#: not in combo
window."); LogState("N ignored"); return; }
        if (activeComboPhase == 1) anyInputPhase1 = true;

        if (comboStage != ComboStage.ExpectMag)
        {
            Debug.Log($"[CubePathMover] WRONG ORDER at N{n} (stage={comboStage})
→ FAIL");
            ResolveCombo(false, reason: $"wrong order (stage={comboStage})");
            return;
        }
        if (n != expectedMagnitude)
        {
            Debug.Log($"[CubePathMover] WRONG MAGNITUDE: expected
N{expectedMagnitude} got N{n} → FAIL");
            ResolveCombo(false, reason: $"wrong magnitude (expected
{expectedMagnitude}, got {n})");
            return;
        }
        comboStage = ComboStage.ExpectRun;
        Debug.Log("[CubePathMover] N" + n + " accepted → expect RUN");
        LogState("after N");
    }

    public void ReportRunPressed()
    {
        Debug.Log("[CubePathMover][INPUT] RUN");
        if (!inComboWindow)
        {
            Debug.Log("[CubePathMover] Ignored RUN: not in combo window.");
```

```
            LogState("RUN ignored");
            return;
        }

        if (comboStage != ComboStage.ExpectRun)
        {
            Debug.Log($"[CubePathMover] RUN at wrong step (stage={comboStage}) → FAIL");
            ResolveCombo(false, reason: $"RUN at wrong step (stage={comboStage})");
            return;
        }

        Debug.Log("[CubePathMover] RUN accepted → SUCCESS");
        ResolveCombo(true, reason: "correct sequence");
    }

    // ----------------- Phase 2 entry (from controller: start_2) -----------------
    public void TriggerStart2Sequence()
    {
        Debug.Log("[CubePathMover] TriggerStart2Sequence()");
        if (!phase2Ready)
        {
            Debug.Log("[CubePathMover] Start_2 pressed but Phase 2 not ready (Text_3 not shown).");
            return;
        }

        StartCoroutine(Start2Pipeline());
    }

    private IEnumerator Start2Pipeline()
    {
        if (start2Delay > 0f)
        {
            Debug.Log($"[CubePathMover] Waiting {start2Delay}s before Phase 2 movement.");
            yield return new WaitForSeconds(start2Delay);
        }

        // Smooth pre-rotation to face Phase 2 direction: (0,0,0)
        Quaternion targetRotP2 = Quaternion.Euler(rotTarget_P2_Euler);
        Debug.Log($"[CubePathMover] Phase 2: Rotating to {rotTarget_P2_Euler} over {rotDuration_P2}s (smoothStep={smoothStep})");
        yield return RotateTo(targetRotP2, rotDuration_P2);

        SafeOn(Text_3);
```

```
        Debug.Log($"[CubePathMover] Phase 2: Moving to r2 over
{moveDuration_P2}s.");
        yield return MoveTo(r2, moveDuration_P2);
        if (dwellAt_r2_P2 > 0f) { yield return new WaitForSeconds(dwellAt_r2_P2); }

        Debug.Log($"[CubePathMover] Phase 2: Moving to r1 over
{moveDuration_P2}s.");
        yield return MoveTo(r1, moveDuration_P2);
        if (dwellAt_r1_P2 > 0f) { yield return new WaitForSeconds(dwellAt_r1_P2); }

        Debug.Log($"[CubePathMover] Phase 2: Moving to pos0 over
{moveDuration_P2}s.");
        yield return MoveTo(pos0, moveDuration_P2);
        if (dwellAt_pos0_P2 > 0f) { yield return new WaitForSeconds(dwellAt_pos0_P2);
}

        Debug.Log($"[CubePathMover] Phase 2: Moving to l1 over
{moveDuration_P2}s.");
        yield return MoveTo(l1, moveDuration_P2);
        if (dwellAt_l1_P2 > 0f) { yield return new WaitForSeconds(dwellAt_l1_P2); }

        Debug.Log($"[CubePathMover] Phase 2: Moving to l2 over
{moveDuration_P2}s.");
        yield return MoveTo(l2, moveDuration_P2);

        SafeOff(Text_3);

        if (appearText4Delay > 0f) { yield return new WaitForSeconds(appearText4Delay);
}
        SafeOn(Text_4);

        Debug.Log("[CubePathMover] Text_4 shown (arrived at l2). Opening P2
combo.");
        OpenComboWindow(ComboDirection.Left, 5, comboWindowSeconds_P2,
phase: 2);
    }

    // ----------------- Optional Reset -----------------
    public void ResetAll()
    {
        if (runRoutine != null) { StopCoroutine(runRoutine); runRoutine = null; }
        transform.position = pos0;
        transform.rotation = Quaternion.Euler(0f, 270f, 0f);

        if (comboRoutine != null) { StopCoroutine(comboRoutine); comboRoutine = null;
}
        comboResolved = false;
```

```
        inComboWindow = false;
        comboStage = ComboStage.None;
        expectedMagnitude = 0;
        activeComboPhase = 0;
        anyInputPhase1 = false;

        SafeOn(Text_1);
        SafeOff(Text_2);
        SafeOff(Text_3);
        SafeOff(Text_4);
        SafeOff(Score_Text);

        score = 0;
        phase2Ready = false;

        Debug.Log("[CubePathMover] ResetAll → pos=pos0, rot=(0,270,0), UI reset,
combo cleared, score=0.");
        LogState("after reset");
    }

    // --- tiny helpers ---
    private void SafeOn(Behaviour b) { if (b != null) b.gameObject.SetActive(true); }
    private void SafeOff(Behaviour b) { if (b != null) b.gameObject.SetActive(false); }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 3

MoveCubeAround15

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

public class MoveCubeAround15 : MonoBehaviour
{
   [Header("Buttons")]
   public GameObject Button2;
   public GameObject Button5;

   [Header("Cube")]
   public GameObject cubeToMove;

   [Header("UI Texts (Assign in Inspector)")]
   public Text xText_UI;
   public Text yText_UI;
   public TMP_Text xText_TMP;
   public TMP_Text yText_TMP;

   private bool button2Pressed = false;
   private bool isMoving = false;
   private Transform cubeTransform;

   void Start()
   {
      cubeTransform = cubeToMove.transform;

      // Hide texts at start
      SetTextsVisible(false);
   }

   void OnCollisionEnter(Collision collisionInfo)
   {
      GameObject other = collisionInfo.gameObject;

      if (Button2 != null &&
         (other == Button2 || other.transform.IsChildOf(Button2.transform)))
      {
         button2Pressed = true;
         return;
      }
```

```
    if (Button5 != null &&
        (other == Button5 || other.transform.IsChildOf(Button5.transform)) &&
        button2Pressed)
    {
        if (!isMoving && cubeTransform != null)
            StartCoroutine(MoveAlongPathToFinalPosition());

        button2Pressed = false;
    }
}

private IEnumerator MoveAlongPathToFinalPosition()
{
    isMoving = true;

    // ✅ Show texts when movement starts
    SetTextsVisible(true);

    List<Vector3> path = new List<Vector3>()
    {
        new Vector3(0f,   0.1f,  0f),
        new Vector3(0.5f, 0.1f,  0f),
        new Vector3(1f,   0.1f,  0f),
        new Vector3(1.5f, 0.1f,  0f),
        new Vector3(2f,   0.1f,  0f),
        new Vector3(2.5f, 0.1f,  0f),
        new Vector3(3f,   0.1f,  0f),
        new Vector3(3.5f, 0.1f,  0f),

        new Vector3(0f,   0.1f, -0.5f),
        new Vector3(0.5f, 0.1f, -0.5f),
        new Vector3(1f,   0.1f, -0.5f),
        new Vector3(1.5f, 0.1f, -0.5f),
        new Vector3(2f,   0.1f, -0.5f),
        new Vector3(2.5f, 0.1f, -0.5f),
        new Vector3(3f,   0.1f, -0.5f),
        new Vector3(3.5f, 0.1f, -0.5f),

        new Vector3(0f,   0.1f, -1f),
        new Vector3(0.5f, 0.1f, -1f),
        new Vector3(1f,   0.1f, -1f),
        new Vector3(1.5f, 0.1f, -1f),
        new Vector3(2f,   0.1f, -1f),
        new Vector3(2.5f, 0.1f, -1f)
    };

    cubeTransform.position = path[0];
```

```
        UpdateCoordinateTexts(path[0]);

        for (int i = 0; i < path.Count; i++)
        {
            cubeTransform.position = path[i];
            UpdateCoordinateTexts(path[i]);

            yield return new WaitForSeconds(0.5f);
        }

        isMoving = false;
    }

    // ✅ Converts world position into your grid numbers
    private void UpdateCoordinateTexts(Vector3 pos)
    {
        int gridX = Mathf.RoundToInt(Mathf.Abs(pos.z) / 0.5f); // row counter
        int gridY = Mathf.RoundToInt(pos.x / 0.5f);          // column counter

        SetTextValues(gridX, gridY);
    }

    private void SetTextValues(int x, int y)
    {
        if (xText_UI != null) xText_UI.text = x.ToString();
        if (yText_UI != null) yText_UI.text = y.ToString();

        if (xText_TMP != null) xText_TMP.text = x.ToString();
        if (yText_TMP != null) yText_TMP.text = y.ToString();
    }

    private void SetTextsVisible(bool visible)
    {
        if (xText_UI != null) xText_UI.gameObject.SetActive(visible);
        if (yText_UI != null) yText_UI.gameObject.SetActive(visible);

        if (xText_TMP != null) xText_TMP.gameObject.SetActive(visible);
        if (yText_TMP != null) yText_TMP.gameObject.SetActive(visible);
    }
}
```

MoveAround15

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

public class MoveAround15 : MonoBehaviour
{
   public GameObject startText;   // Drag your Text or TMP object here in the
Inspector

   void Start()
   {
      // Move object to (0, 0.1, 0)
      transform.position = new Vector3(0f, 0.1f, 0f);

      // Show text at the start of the scene
      if (startText != null)
      {
         startText.SetActive(true);
      }
      else
      {
         Debug.LogWarning("Start Text is not assigned in the Inspector!");
      }
   }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 4

MoveCubeAround16

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

public class MoveCubeAround16 : MonoBehaviour
{
    [Header("Buttons Phase 1")]
    public GameObject Start_1;        // Assign your Start_1 button here
    public GameObject Button_3_1;     // First Button 3 (for sequence 1)
    public GameObject Button_4;       // Button 4 (for sequence 1)

    [Header("Buttons Phase 2")]
    public GameObject Start_2;        // Assign Start_2 button here
    public GameObject Button_1;       // Button 1 (for sequence 2)
    public GameObject Button_3_2;     // Second Button 3 (for sequence 2)

    [Header("Cube")]
    public GameObject cubeToMove;     // Assign the cube you want to move

    [Header("UI")]
    public GameObject text_1;         // Assign your Text_1 (UI object) here
    public GameObject text_2;         // Assign your Text_2 (UI object) here
    public GameObject text_3;         // Assign your Text_3 (UI object) here
    public GameObject text_4;         // Assign your Text_4 (UI object) here
    public TMP_Text ScoreText;        // Assign your ScoreText TMP Text here

    [Header("Audio")]
    public AudioSource successSound;   // Assign an AudioSource with the success
sound

    private Transform cubeTransform;
    private bool hasStarted = false;
    private bool isMoving = false;

    // How long the smooth transitions should take (seconds)
    public float smoothDuration = 1f;

    // How long the cube stays at each waypoint (seconds)
    public float waitPerPoint = 0.5f;

    // Path waypoints (phase 1)
    private List<Vector3> pathPoints;
```

```
    // Indices in the path that should have a smooth transition from the previous point
(phase 1)
    private HashSet<int> smoothIndices;

    // Button sequence 1 logic (Buttons 3_1 -> 4 with timer)
    private bool readyForSequence = false;
    private bool button3Pressed = false;
    private bool sequenceCompleted = false;
    private Coroutine timerCoroutine;
    private float sequenceTimeLimit = 7f;   // 7 seconds for button 3_1 then 4

    // Phase 2 movement and logic
    private bool phase2Ready = false;
    private bool hasStartedPhase2 = false;

    // Button sequence 2 logic (Buttons 1 -> 3_2, no timer)
    private bool readyForSecondSequence = false;
    private bool button1Pressed = false;
    private bool secondSequenceCompleted = false;

    // Score value
    private int currentScore = 0;

    void Start()
    {
        if (cubeToMove != null)
            cubeTransform = cubeToMove.transform;

        // Show text_1 at the start of the scene, hide others
        if (text_1 != null)
            text_1.SetActive(true);
        if (text_2 != null)
            text_2.SetActive(false);
        if (text_3 != null)
            text_3.SetActive(false);
        if (text_4 != null)
            text_4.SetActive(false);

        // Clear ScoreText at start
        currentScore = 0;
        if (ScoreText != null)
            ScoreText.text = "";

        // Build the path for phase 1
        pathPoints = new List<Vector3>()
        {
            // z = 0
```

```
        new Vector3(0f,   0.1f,  0f),   // index 0
        new Vector3(0.5f, 0.1f,  0f),   // 1
        new Vector3(1f,   0.1f,  0f),   // 2
        new Vector3(1.5f, 0.1f,  0f),   // 3
        new Vector3(2f,   0.1f,  0f),   // 4
        new Vector3(2.5f, 0.1f,  0f),   // 5
        new Vector3(3f,   0.1f,  0f),   // 6
        new Vector3(3.5f, 0.1f,  0f),   // 7

        // z = -0.5
        new Vector3(0f,   0.1f, -0.5f), // 8   (smooth from 7 -> 8)
        new Vector3(0.5f, 0.1f, -0.5f), // 9
        new Vector3(1f,   0.1f, -0.5f), // 10
        new Vector3(1.5f, 0.1f, -0.5f), // 11
        new Vector3(2f,   0.1f, -0.5f), // 12
        new Vector3(2.5f, 0.1f, -0.5f), // 13
        new Vector3(3f,   0.1f, -0.5f), // 14
        new Vector3(3.5f, 0.1f, -0.5f), // 15

        // z = -1
        new Vector3(0f,   0.1f, -1f),   // 16 (smooth from 15 -> 16)
        new Vector3(0.5f, 0.1f, -1f),   // 17
        new Vector3(1f,   0.1f, -1f),   // 18
        new Vector3(1.5f, 0.1f, -1f),   // 19
        new Vector3(2f,   0.1f, -1f),   // 20
        new Vector3(2.5f, 0.1f, -1f),   // 21
        new Vector3(3f,   0.1f, -1f),   // 22
        new Vector3(3.5f, 0.1f, -1f),   // 23

        // z = -1.5
        new Vector3(0f,   0.1f, -1.5f), // 24 (smooth from 23 -> 24)
        new Vector3(0.5f, 0.1f, -1.5f), // 25
        new Vector3(1f,   0.1f, -1.5f), // 26
        new Vector3(1.5f, 0.1f, -1.5f), // 27
        new Vector3(2f,   0.1f, -1.5f), // 28 FINAL
    };

    // These indices will be smoothed *from* their previous point (phase 1)
    // 7 -> 8, 15 -> 16, 23 -> 24
    smoothIndices = new HashSet<int>() { 8, 16, 24 };
}

void OnCollisionEnter(Collision collisionInfo)
{
    GameObject other = collisionInfo.gameObject;

    // Start_1 logic (start movement of phase 1)
```

```
if (cubeTransform != null && !isMoving && !hasStarted)
{
   if (Start_1 != null &&
      (other == Start_1 || other.transform.IsChildOf(Start_1.transform)))
   {
      hasStarted = true;
      StartCoroutine(MoveCubeAlongPath());
      return;
   }
}

// Start_2 logic (start movement of phase 2)
if (cubeTransform != null && !isMoving && phase2Ready && !hasStartedPhase2)
{
   if (Start_2 != null &&
      (other == Start_2 || other.transform.IsChildOf(Start_2.transform)))
   {
      hasStartedPhase2 = true;
      StartCoroutine(MoveCubePhase2());
      return;
   }
}

// Button 3_1 and 4 logic after cube has reached final position of phase 1
if (readyForSequence && !sequenceCompleted)
{
   if (Button_3_1 != null &&
      (other == Button_3_1 || other.transform.IsChildOf(Button_3_1.transform)))
   {
      // First required button
      if (!button3Pressed)
      {
         button3Pressed = true;
      }
   }
   else if (Button_4 != null &&
         (other == Button_4 || other.transform.IsChildOf(Button_4.transform)))
   {
      // Second required button, only counts if Button 3_1 was already pressed
      if (button3Pressed)
      {
         sequenceCompleted = true;
         readyForSequence = false;

         // Stop timer and give score
         if (timerCoroutine != null)
         {
```

```
                StopCoroutine(timerCoroutine);
                timerCoroutine = null;
            }

            currentScore = 10;
            if (ScoreText != null)
                ScoreText.text = "Score: 10/20";

            PlaySuccessSound();

            // Move on to phase 2 (texts & Start_2)
            BeginPhase2();
        }
    }
}

// Second sequence: Button 1 then Button 3_2 (after phase 2 movement)
if (readyForSecondSequence && !secondSequenceCompleted)
{
    if (Button_1 != null &&
        (other == Button_1 || other.transform.IsChildOf(Button_1.transform)))
    {
        // First required button
        if (!button1Pressed)
        {
            button1Pressed = true;
        }
    }
    else if (Button_3_2 != null &&
             (other == Button_3_2 ||
other.transform.IsChildOf(Button_3_2.transform)))
    {
        // Second required button, only counts if Button 1 was already pressed
        if (button1Pressed)
        {
            secondSequenceCompleted = true;
            readyForSecondSequence = false;

            // Update score depending on previous score (+10 if success)
            currentScore += 10;
            if (currentScore > 20) currentScore = 20;

            if (ScoreText != null)
                ScoreText.text = "Score: " + currentScore + "/20";

            PlaySuccessSound();
        }
```

```
        }
    }
}

private IEnumerator MoveCubeAlongPath()
{
    isMoving = true;

    // *** 3 second delay before movement starts (after Start_1 is pressed) ***
    yield return new WaitForSeconds(3f);

    if (pathPoints == null || pathPoints.Count == 0)
    {
        isMoving = false;
        yield break;
    }

    // Snap to first waypoint and wait
    cubeTransform.position = pathPoints[0];
    yield return new WaitForSeconds(waitPerPoint);

    // Go through all waypoints (phase 1)
    for (int i = 1; i < pathPoints.Count; i++)
    {
        Vector3 start = cubeTransform.position;
        Vector3 end = pathPoints[i];

        bool useSmooth = smoothIndices.Contains(i);

        if (useSmooth)
        {
            float elapsed = 0f;
            while (elapsed < smoothDuration)
            {
                elapsed += Time.deltaTime;
                float t = Mathf.Clamp01(elapsed / smoothDuration);
                cubeTransform.position = Vector3.Lerp(start, end, t);
                yield return null;
            }
            cubeTransform.position = end; // ensure exact
        }
        else
        {
            // Instant move
            cubeTransform.position = end;
        }
```

```
        // Stay at this point
        yield return new WaitForSeconds(waitPerPoint);
    }

    isMoving = false;

    // Path complete, cube is at final position (approximately 2, 0.1, -1.5)
    OnPathComplete();
}

private void OnPathComplete()
{
    // Switch texts: hide text_1, show text_2
    if (text_1 != null)
        text_1.SetActive(false);
    if (text_2 != null)
        text_2.SetActive(true);

    // Prepare for button 3_1 then 4 sequence
    readyForSequence = true;
    button3Pressed = false;
    sequenceCompleted = false;

    // Start 7-second timer
    if (timerCoroutine != null)
    {
        StopCoroutine(timerCoroutine);
    }
    timerCoroutine = StartCoroutine(SequenceTimer());
}

private IEnumerator SequenceTimer()
{
    float elapsed = 0f;

    while (elapsed < sequenceTimeLimit && !sequenceCompleted)
    {
        elapsed += Time.deltaTime;
        yield return null;
    }

    // If time ran out and sequence not completed, score is 0
    if (!sequenceCompleted)
    {
        readyForSequence = false;

        currentScore = 0;
```

```
            if (ScoreText != null)
                ScoreText.text = "Score: 0/20";

            // Move on to phase 2 as well
            BeginPhase2();
        }

        timerCoroutine = null;
    }

    private void BeginPhase2()
    {
        if (phase2Ready) return; // avoid running twice

        phase2Ready = true;

        // When user has pressed 3_1 & 4 OR time has passed:
        // text_2 disappears, text_3 appears
        if (text_2 != null)
            text_2.SetActive(false);
        if (text_3 != null)
            text_3.SetActive(true);
    }

    private IEnumerator MoveCubePhase2()
    {
        isMoving = true;

        // Phase 2: start position is (0, 0, 0)
        cubeTransform.position = new Vector3(0f, 0f, 0f);

        // *** 3 second delay before second phase movement starts (after Start_2 is
pressed) ***
        yield return new WaitForSeconds(3f);

        // Define phase 2 path
        List<Vector3> phase2Path = new List<Vector3>()
        {
            // z = 0 row
            new Vector3(0f,   0.1f,  0f),  // 0
            new Vector3(0.5f, 0.1f,  0f),  // 1
            new Vector3(1f,   0.1f,  0f),  // 2
            new Vector3(1.5f, 0.1f,  0f),  // 3
            new Vector3(2f,   0.1f,  0f),  // 4
            new Vector3(2.5f, 0.1f,  0f),  // 5
            new Vector3(3f,   0.1f,  0f),  // 6
            new Vector3(3.5f, 0.1f,  0f),  // 7
```

```
    // z = -0.5 row
    new Vector3(0f,   0.1f, -0.5f), // 8 (smooth from 7 -> 8)
    new Vector3(0.5f, 0.1f, -0.5f), // 9
    new Vector3(1f,   0.1f, -0.5f), // 10
    new Vector3(1.5f, 0.1f, -0.5f), // 11 FINAL (for text_4)
};

// Only index 8 (0,0.1,-0.5) is reached via smooth transition from 7 (3.5,0.1,0)
HashSet<int> phase2SmoothIndices = new HashSet<int>() { 8 };

// Snap to first waypoint and wait
cubeTransform.position = phase2Path[0];
yield return new WaitForSeconds(waitPerPoint);

for (int i = 1; i < phase2Path.Count; i++)
{
    Vector3 start = cubeTransform.position;
    Vector3 end = phase2Path[i];

    bool useSmooth = phase2SmoothIndices.Contains(i);

    if (useSmooth)
    {
        float elapsed = 0f;
        while (elapsed < smoothDuration)
        {
            elapsed += Time.deltaTime;
            float t = Mathf.Clamp01(elapsed / smoothDuration);
            cubeTransform.position = Vector3.Lerp(start, end, t);
            yield return null;
        }
        cubeTransform.position = end;
    }
    else
    {
        // Instant move
        cubeTransform.position = end;
    }

    // Stay at this point for 0.5s
    yield return new WaitForSeconds(waitPerPoint);
}

isMoving = false;

OnPhase2Complete();
```

```
    }

    private void OnPhase2Complete()
    {
        // When cube is at (1.5, 0.1, -0.5):
        // text_4 appears, text_3 disappears
        if (text_3 != null)
            text_3.SetActive(false);
        if (text_4 != null)
            text_4.SetActive(true);

        // Now user must press buttons 1 and 3_2 (in this order)
        readyForSecondSequence = true;
        button1Pressed = false;
        secondSequenceCompleted = false;
    }

    private void PlaySuccessSound()
    {
        if (successSound != null)
        {
            successSound.Play();
        }
    }
}
```

MoveAround16

```
using UnityEngine;

public class MoveAround16 : MonoBehaviour
{
   void Start()
   {
      // Set the cube's starting position
      transform.position = new Vector3(0f, 0.1f, 0f);
   }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 5

PinocchioMovement17

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

public class PinocchioMovement17 : MonoBehaviour
{
    [Header("Movement")]
    public Vector3 startPosition = new Vector3(18.5f, 2f, 88f);
    public Vector3 endPosition = new Vector3(-19f, 2f, 88f);

    [Tooltip("Reference to the LightToggle17 in the scene.")]
    public LightToggle17 lightToggle;

    [Header("Arrival Behavior")]
    [Tooltip("How long Pinocchio stays at the end before disappearing.")]
    public float endHoldSeconds = 2f;

    [Header("Movement Durations")]
    [Tooltip("Seconds to travel from start to end for the FIRST run.")]
    public float firstMoveDuration = 10f;

    [Tooltip("Seconds to travel from start to end for the SECOND and later runs.")]
    public float secondMoveDuration = 5f;

    // Internal state
    private float t;
    private bool moving;
    private bool firstRunStarted;
    private bool waitingForCountdown7;
    private int lastSeenCountdown = int.MinValue;

    private float currentMoveDuration;

    // NEW: run tracking
    private int completedRuns = 0;
    private bool finishedAllRuns = false;

    // Renderers to toggle visibility without disabling the GameObject
    private List<Renderer> renderers = new List<Renderer>();

    private void Awake()
    {
        transform.position = startPosition;
```

```
    CacheRenderers();
}

private void OnEnable()
{
    if (lightToggle != null)
    {
        lightToggle.OnGreenStarted += HandleGreenStarted;
        lightToggle.OnRedStarted += HandleRedStarted;
    }
}

private void OnDisable()
{
    if (lightToggle != null)
    {
        lightToggle.OnGreenStarted -= HandleGreenStarted;
        lightToggle.OnRedStarted -= HandleRedStarted;
    }
}

private void Update()
{
    if (lightToggle == null || finishedAllRuns) return;

    // First run: wait for GREEN and countdown == 13
    if (!firstRunStarted)
    {
        int c = lightToggle.GetCurrentCountdown();
        if (lightToggle.IsGreenLight() && c != lastSeenCountdown && c == 13)
        {
            firstRunStarted = true;
            currentMoveDuration = Mathf.Max(0.0001f, firstMoveDuration);
            BeginMoveFromStart();
            completedRuns = 1; // mark we're doing the first run
        }
        lastSeenCountdown = c;
    }
    // After reset: wait for GREEN and countdown == 7 (second run)
    else if (waitingForCountdown7)
    {
        int c = lightToggle.GetCurrentCountdown();
        if (lightToggle.IsGreenLight() && c != lastSeenCountdown && c == 7)
        {
            currentMoveDuration = Mathf.Max(0.0001f, secondMoveDuration);
            BeginMoveFromStart();
            completedRuns = 2; // mark we're doing the second run
```

```
        }
        lastSeenCountdown = c;
    }

    // Advance continuously once started
    if (moving)
    {
        t += Time.deltaTime / currentMoveDuration;
        t = Mathf.Clamp01(t);
        transform.position = Vector3.Lerp(startPosition, endPosition, t);

        if (t >= 1f - 1e-5f)
        {
            StartCoroutine(HandleArrivalAtEnd());
        }
    }
}

private void HandleGreenStarted()
{
    // Starts are handled via countdown checks in Update.
}

private void HandleRedStarted()
{
    // Movement continues once started.
}

private void BeginMoveFromStart()
{
    t = 0f;
    transform.position = startPosition;
    Show(true);

    moving = true;
    waitingForCountdown7 = false;
}

private IEnumerator HandleArrivalAtEnd()
{
    moving = false;
    transform.position = endPosition;

    // If we've completed the second movement, stop here permanently.
    if (completedRuns >= 2)
    {
        finishedAllRuns = true;
```

```
            yield break; // do NOT reset; stay at (-19, 2, 88)
        }

        // Otherwise (after first run), do your existing reset flow and prepare for second
run
        yield return new WaitForSeconds(endHoldSeconds);

        Show(false);
        transform.position = startPosition;
        yield return null;
        Show(true);

        // Next run starts at GREEN + countdown == 7
        waitingForCountdown7 = true;
    }

    private void CacheRenderers()
    {
        renderers.Clear();
        GetComponentsInChildren(true, renderers);
    }

    private void Show(bool visible)
    {
        foreach (var r in renderers)
        {
            if (r != null) r.enabled = visible;
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 6

CollisionHandler

```
using UnityEngine;

public class CollisionHandler : MonoBehaviour
{
   public PinocchioMovement targetScript;  // Script that contains the movement
logic
   public GameObject Button_walk;
   public GameObject Button_run;


   void OnCollisionEnter(Collision collisionInfo)
   {
      if (collisionInfo.collider.name == "Cube_Button1")
      {
         targetScript.StartWalking();
      }
      else if (collisionInfo.collider.name == "Cube_Button3")
      {
         targetScript.StartRunning();
      }
   }
}
```

PinocchioMovement

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI; // Include the TextMeshPro namespace


public class PinocchioMovement : MonoBehaviour
{
   public float walkDuration;
   public float runDuration;
   private Vector3 startPosition = new Vector3(0, 0, 0);
   private Vector3 endPosition = new Vector3(6, 0, 0);
   private bool isMoving = false;
   private float startTime;
   private float speed;

   public LightToggle lightToggle; // Assigned in the inspector
   public AudioSource audioSource; // Assigned an AudioSource component
   public AudioClip redLightSound; // Assign clips for each condition
   public Text scoreText; // Assign this in the inspector


   void Start()
   {
      transform.position = startPosition;
      scoreText.text = "Score: 0/10"; // Initialize the score text at the start
      scoreText.gameObject.SetActive(true); // Show the score text at start
   }

   void Update()
   {
      if (isMoving)
      {
         float timeSinceStarted = Time.time - startTime;
         float fractionOfJourney = timeSinceStarted / speed;
         transform.position = Vector3.Lerp(startPosition, endPosition,
fractionOfJourney);

         // Check if Pinocchio has reached the end position
         if (Vector3.Distance(transform.position, endPosition) < 0.01f)
         {
            isMoving = false;
         }
      }
   }

   public void StartWalking()
```

```
    {
        StartMoving(walkDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() >= 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
        }
        else
        {
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    public void StartRunning()
    {
        StartMoving(runDuration);
        if (lightToggle.IsGreenLight())
        {
            if (lightToggle.GetCurrentCountdown() < 5)
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
            }
            else if (lightToggle.GetCurrentCountdown() >= 5 &&
lightToggle.GetCurrentCountdown() < 10)
            {
                UpdateScore(true); // Score 10/10 and no sound played
            }
            // Modified condition to update the score to 0/10 when the counter is
between 15 and 10
            else if (lightToggle.GetCurrentCountdown() >= 10 &&
lightToggle.GetCurrentCountdown() <= 15)
            {
                UpdateScore(false); // Update score to 0/10
            }
            else
            {
                UpdateScore(false);
                StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
```

```
            }
        }
        else
        {
            UpdateScore(false);
            StartCoroutine(PlaySoundWithDelay(redLightSound, 1.0f));
        }
    }

    private void StartMoving(float duration)
    {
        if (!isMoving)
        {
            isMoving = true;
            startTime = Time.time;
            speed = 1.0f / duration;
        }
    }

    private IEnumerator PlaySoundWithDelay(AudioClip clip, float delay)
    {
        yield return new WaitForSeconds(delay);
        audioSource.PlayOneShot(clip);
    }

    private void UpdateScore(bool conditionMet)
    {
        if (conditionMet)
            scoreText.text = "Score: 10/10";
        else
            scoreText.text = "Score: 0/10";
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 7

ButtonsInteraction19

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

public class ButtonsInteraction19 : MonoBehaviour
{
    [Header("Buttons (drag from Hierarchy)")]
    public GameObject button2;
    public GameObject button3;
    public GameObject button4;

    [Header("Texts (drag from Hierarchy)")]
    public GameObject text1;
    public GameObject text2;
    public GameObject text3;
    public GameObject text4;

    [Header("N Text Object (drag the GameObject here)")]
    public GameObject nTextObject;
    public bool showNText = true;

    [Header("Timing: Texts")]
    public float delaySecondsTexts = 5f;

    [Header("Cubes (drag from Hierarchy)")]
    public GameObject cube1;
    public GameObject cube2;
    public GameObject cube3;
    public GameObject cube4;
    public GameObject cube5;
    public GameObject cube6;
    public GameObject cube7;
    public GameObject cube8;
    public GameObject cube9;
    public GameObject cube10;

    [Header("Cube timing (ONE value for ALL block delays)")]
    [Tooltip("Used for every delay between cube blocks (all the '1 sec later' steps).")]
    public float cubeBlockDelaySeconds = 1f;

    [Header("Extra cube timing")]
    [Tooltip("Delay after the last visible text appears (Text1) before Cube1 appears.")]
```

```
    public float delayAfterLastTextBeforeCube1 = 5f;

    private bool touchingB2, touchingB3, touchingB4;
    private Coroutine running;

    private TMP_Text nTmp;
    private Text nUi;

    private void Awake()
    {
        CacheNTextComponent();
        HideAllTexts();
        HideAllCubes();
        SetN(null);
    }

    private void CacheNTextComponent()
    {
        nTmp = null;
        nUi = null;

        if (nTextObject == null) return;

        nTmp = nTextObject.GetComponent<TMP_Text>();
        if (nTmp != null) return;

        nUi = nTextObject.GetComponent<Text>();
    }

    private void SetN(int? n)
    {
        if (nTextObject == null) return;

        if (!showNText || n == null)
        {
            nTextObject.SetActive(false);
            return;
        }

        nTextObject.SetActive(true);
        string value = $"N={n.Value}";

        if (nTmp != null) nTmp.text = value;
        else if (nUi != null) nUi.text = value;
        else Debug.LogWarning("nTextObject has no TMP_Text or UnityEngine.UI.Text
component.");
    }
```

```
    private void HideAllTexts()
    {
        if (text1 != null) text1.SetActive(false);
        if (text2 != null) text2.SetActive(false);
        if (text3 != null) text3.SetActive(false);
        if (text4 != null) text4.SetActive(false);
    }

    private void HideAllCubes()
    {
        if (cube1 != null) cube1.SetActive(false);
        if (cube2 != null) cube2.SetActive(false);
        if (cube3 != null) cube3.SetActive(false);
        if (cube4 != null) cube4.SetActive(false);
        if (cube5 != null) cube5.SetActive(false);
        if (cube6 != null) cube6.SetActive(false);
        if (cube7 != null) cube7.SetActive(false);
        if (cube8 != null) cube8.SetActive(false);
        if (cube9 != null) cube9.SetActive(false);
        if (cube10 != null) cube10.SetActive(false);
    }

    private float D() => Mathf.Max(0f, cubeBlockDelaySeconds);
    private float StartDelay() => Mathf.Max(0f, delayAfterLastTextBeforeCube1);

    private void OnTriggerEnter(Collider other) => HandleEnter(other.transform);
    private void OnTriggerExit(Collider other) => HandleExit(other.transform);

    private void OnCollisionEnter(Collision collision) =>
HandleEnter(collision.transform);
    private void OnCollisionExit(Collision collision) => HandleExit(collision.transform);

    private void HandleEnter(Transform other)
    {
        if (button4 != null && IsPartOf(other, button4.transform))
        {
            if (!touchingB4)
            {
                touchingB4 = true;
                SetN(4);
                StartRound(startText: 4, maxCube: 10);
            }
            return;
        }

        if (button3 != null && IsPartOf(other, button3.transform))
```

```
        {
            if (!touchingB3)
            {
                touchingB3 = true;
                SetN(3);
                StartRound(startText: 3, maxCube: 6);
            }
            return;
        }

        if (button2 != null && IsPartOf(other, button2.transform))
        {
            if (!touchingB2)
            {
                touchingB2 = true;
                SetN(2);
                StartRound(startText: 2, maxCube: 3);
            }
            return;
        }
    }

    private void HandleExit(Transform other)
    {
        if (button4 != null && IsPartOf(other, button4.transform)) touchingB4 = false;
        if (button3 != null && IsPartOf(other, button3.transform)) touchingB3 = false;
        if (button2 != null && IsPartOf(other, button2.transform)) touchingB2 = false;
    }

    private bool IsPartOf(Transform hit, Transform buttonRoot)
    {
        return hit == buttonRoot || hit.IsChildOf(buttonRoot);
    }

    private void StartRound(int startText, int maxCube)
    {
        if (running != null) StopCoroutine(running);

        // Start new round: clear previous visuals (this also clears the "last cubes" from
previous round)
        HideAllTexts();
        HideAllCubes();

        running = StartCoroutine(TextsThenCubes(startText, maxCube));
    }

    private IEnumerator TextsThenCubes(int startText, int maxCube)
```

```
  {
    // Show starting text immediately
    if (startText == 4 && text4 != null) text4.SetActive(true);
    if (startText == 3 && text3 != null) text3.SetActive(true);
    if (startText == 2 && text2 != null) text2.SetActive(true);
    if (startText == 1 && text1 != null) text1.SetActive(true);

    // Reveal remaining texts downwards every delaySecondsTexts
    if (startText >= 4)
    {
      yield return new WaitForSeconds(delaySecondsTexts);
      if (text3 != null) text3.SetActive(true);
    }

    if (startText >= 3)
    {
      yield return new WaitForSeconds(delaySecondsTexts);
      if (text2 != null) text2.SetActive(true);
    }

    if (startText >= 2)
    {
      yield return new WaitForSeconds(delaySecondsTexts);
      if (text1 != null) text1.SetActive(true);
    }

    // After last visible text (Text1), wait then start cubes
    yield return new WaitForSeconds(StartDelay());
    yield return StartCoroutine(CubeTimeline(maxCube));

    // IMPORTANT: Do NOT hide cubes at the end.
    // The last cubes remain visible until another button is pressed (StartRound
clears them).
    running = null;
  }

  private IEnumerator CubeTimeline(int maxCube)
  {
    float d = D();

    // Cube1 appears then disappears
    if (maxCube >= 1)
    {
      if (cube1 != null) cube1.SetActive(true);
      yield return new WaitForSeconds(d);
      if (cube1 != null) cube1.SetActive(false);
    }
```

```
else yield break;

// Cube2 appears
if (maxCube >= 2)
{
   if (cube2 != null) cube2.SetActive(true);
}
else yield break;

// 1 block later Cube3 appears
yield return new WaitForSeconds(d);
if (maxCube >= 3)
{
   if (cube3 != null) cube3.SetActive(true);

   // If stopping at Cube3, keep Cube2 & Cube3 visible
   if (maxCube <= 3) yield break;
}
else yield break;

// 1 block later Cube2 & Cube3 disappear together
yield return new WaitForSeconds(d);
if (cube2 != null) cube2.SetActive(false);
if (cube3 != null) cube3.SetActive(false);

// 1 block later Cubes4 & 5 appear together
yield return new WaitForSeconds(d);
if (maxCube >= 4)
{
   if (cube4 != null) cube4.SetActive(true);
   if (cube5 != null) cube5.SetActive(true);
}
else yield break;

// 1 block later Cube6 appears
yield return new WaitForSeconds(d);
if (maxCube >= 6)
{
   if (cube6 != null) cube6.SetActive(true);

   // If stopping at Cube6, keep Cubes4,5,6 visible
   if (maxCube <= 6) yield break;
}
else yield break;

// 1 block later Cubes4,5,6 disappear together
yield return new WaitForSeconds(d);
```

```
        if (cube4 != null) cube4.SetActive(false);
        if (cube5 != null) cube5.SetActive(false);
        if (cube6 != null) cube6.SetActive(false);

        // 1 block later Cubes7,8,9 appear together
        yield return new WaitForSeconds(d);
        if (maxCube >= 7)
        {
            if (cube7 != null) cube7.SetActive(true);
            if (cube8 != null) cube8.SetActive(true);
            if (cube9 != null) cube9.SetActive(true);
        }
        else yield break;

        // 1 block later Cube10 appears (and remains)
        yield return new WaitForSeconds(d);
        if (maxCube >= 10)
        {
            if (cube10 != null) cube10.SetActive(true);
        }
    }
}
```

ΚΩΔΙΚΑΣ ΣΚΗΝΗ 8

ButtonsInteraction20

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

public class ButtonsInteraction20 : MonoBehaviour
{
    [Header("Start Buttons (drag from Hierarchy)")]
    public GameObject start1;
    public GameObject start2;

    [Header("Answer Buttons (drag from Hierarchy)")]
    public GameObject button2;
    public GameObject button3;

    [Header("Instruction Texts (drag from Hierarchy)")]
    public GameObject instruction_text_1;
    public GameObject instruction_text_2;

    [Header("Answer Texts (drag from Hierarchy)")]
    public GameObject answer1;
    public GameObject answer2;
    public GameObject answer3;

    [Header("Score Text (drag the GameObject here)")]
    public GameObject scoreTextObject;

    [Header("Sounds")]
    public AudioSource audioSource;
    public AudioClip correctPhase1Clip; // when Button3 is pressed in phase 1
    public AudioClip correctPhase2Clip; // when Button2 is pressed in phase 2

    [Header("Cubes (drag from Hierarchy)")]
    public GameObject cube1;
    public GameObject cube2;
    public GameObject cube3;
    public GameObject cube4;
    public GameObject cube5;
    public GameObject cube6;

    [Header("Timing: Cubes")]
    public float cubeBlockDelaySeconds = 1f;
    public float delayBeforeCube1 = 5f;
```

```
[Header("Extra Cube Time")]
[Tooltip("Adds +3 seconds to every cube appear/disappear delay.")]
public float extraCubeSeconds = 3f;

[Header("Timing: Instructions -> Cubes")]
public float delayAfterInstruction2BeforeCubes = 5f;

[Header("Timing: Score Windows")]
public float scoreWindowButton3Seconds = 12f;
public float scoreWindowButton2Seconds = 10f;

private bool touchingStart1, touchingStart2;
private bool touchingBtn2, touchingBtn3;

private Coroutine flowRoutine;
private Coroutine scoreRoutine;

private TMP_Text scoreTMP;
private Text scoreUI;

private enum ScoreStage { None, WaitingForButton3, WaitingForButton2 }
private ScoreStage scoreStage = ScoreStage.None;

private bool firstScoreShown = false;
private int currentScore = 0;

private void Awake()
{
    CacheScoreComponent();

    if (instruction_text_1 != null) instruction_text_1.SetActive(true);
    if (instruction_text_2 != null) instruction_text_2.SetActive(false);

    SetAnswersVisible(false);

    HideScore();
    HideAllCubes();
}

private void CacheScoreComponent()
{
    scoreTMP = null;
    scoreUI = null;

    if (scoreTextObject == null) return;
```

```
        scoreTMP = scoreTextObject.GetComponent<TMP_Text>();
        if (scoreTMP != null) return;

        scoreUI = scoreTextObject.GetComponent<Text>();
    }

    private void SetAnswersVisible(bool visible)
    {
        if (answer1 != null) answer1.SetActive(visible);
        if (answer2 != null) answer2.SetActive(visible);
        if (answer3 != null) answer3.SetActive(visible);
    }

    private void HideScore()
    {
        if (scoreTextObject != null) scoreTextObject.SetActive(false);
    }

    private void UpdateScoreUI()
    {
        if (scoreTextObject == null) return;

        scoreTextObject.SetActive(true);
        string value = $"Score: {currentScore}/20 ";

        if (scoreTMP != null) scoreTMP.text = value;
        else if (scoreUI != null) scoreUI.text = value;
        else Debug.LogWarning("ScoreTextObject has no TMP_Text or
UnityEngine.UI.Text component.");

        if (!firstScoreShown)
        {
            firstScoreShown = true;
            if (instruction_text_1 != null) instruction_text_1.SetActive(false);
            if (instruction_text_2 != null) instruction_text_2.SetActive(false);
        }
    }

    private void HideAllCubes()
    {
        if (cube1 != null) cube1.SetActive(false);
        if (cube2 != null) cube2.SetActive(false);
        if (cube3 != null) cube3.SetActive(false);
        if (cube4 != null) cube4.SetActive(false);
        if (cube5 != null) cube5.SetActive(false);
        if (cube6 != null) cube6.SetActive(false);
    }
```

```
    private float D() => Mathf.Max(0f, cubeBlockDelaySeconds + extraCubeSeconds);
    private float StartDelay() => Mathf.Max(0f, delayBeforeCube1);

    private void OnTriggerEnter(Collider other) => HandleEnter(other.transform);
    private void OnTriggerExit(Collider other) => HandleExit(other.transform);

    private void OnCollisionEnter(Collision collision) =>
HandleEnter(collision.transform);
    private void OnCollisionExit(Collision collision) => HandleExit(collision.transform);

    private void HandleEnter(Transform other)
    {
        if (start1 != null && IsPartOf(other, start1.transform))
        {
            if (!touchingStart1)
            {
                touchingStart1 = true;
                StartMainFlow();
            }
            return;
        }

        if (start2 != null && IsPartOf(other, start2.transform))
        {
            if (!touchingStart2)
            {
                touchingStart2 = true;
                SetAnswersVisible(true);
            }
            return;
        }

        if (button3 != null && IsPartOf(other, button3.transform))
        {
            if (!touchingBtn3)
            {
                touchingBtn3 = true;
                OnAnswerButtonPressed(pressedButton2: false, pressedButton3: true);
            }
            return;
        }

        if (button2 != null && IsPartOf(other, button2.transform))
        {
            if (!touchingBtn2)
            {
```

```
            touchingBtn2 = true;
            OnAnswerButtonPressed(pressedButton2: true, pressedButton3: false);
        }
        return;
    }
}

private void HandleExit(Transform other)
{
    if (start1 != null && IsPartOf(other, start1.transform)) touchingStart1 = false;
    if (start2 != null && IsPartOf(other, start2.transform)) touchingStart2 = false;

    if (button2 != null && IsPartOf(other, button2.transform)) touchingBtn2 = false;
    if (button3 != null && IsPartOf(other, button3.transform)) touchingBtn3 = false;
}

private bool IsPartOf(Transform hit, Transform buttonRoot)
{
    return hit == buttonRoot || hit.IsChildOf(buttonRoot);
}

private void StartMainFlow()
{
    if (flowRoutine != null) StopCoroutine(flowRoutine);
    StopScoreWindow();

    HideAllCubes();

    if (instruction_text_1 != null) instruction_text_1.SetActive(true);
    if (instruction_text_2 != null) instruction_text_2.SetActive(false);

    HideScore();

    firstScoreShown = false;
    currentScore = 0;
    scoreStage = ScoreStage.None;

    flowRoutine = StartCoroutine(MainFlowCoroutine());
}

private IEnumerator MainFlowCoroutine()
{
    yield return StartCoroutine(CubeRound1ThenScore());

    if (instruction_text_2 != null) instruction_text_2.SetActive(true);
```

```
        yield return new WaitForSeconds(Mathf.Max(0f,
delayAfterInstruction2BeforeCubes));

        HideAllCubes();

        yield return StartCoroutine(CubeRound2ThenScore());

        flowRoutine = null;
    }

    private IEnumerator CubeRound1ThenScore()
    {
        yield return StartCoroutine(CubesTo6());

        scoreStage = ScoreStage.WaitingForButton3;
        StartScoreWindow(Mathf.Max(0f, scoreWindowButton3Seconds));

        while (scoreStage != ScoreStage.None)
            yield return null;
    }

    private IEnumerator CubeRound2ThenScore()
    {
        yield return StartCoroutine(CubesTo6());

        scoreStage = ScoreStage.WaitingForButton2;
        StartScoreWindow(Mathf.Max(0f, scoreWindowButton2Seconds));

        while (scoreStage != ScoreStage.None)
            yield return null;
    }

    private IEnumerator CubesTo6()
    {
        float d = D();

        yield return new WaitForSeconds(StartDelay());

        if (cube1 != null) cube1.SetActive(true);
        yield return new WaitForSeconds(d);
        if (cube1 != null) cube1.SetActive(false);

        if (cube2 != null) cube2.SetActive(true);

        yield return new WaitForSeconds(d);
        if (cube3 != null) cube3.SetActive(true);
```

```
    yield return new WaitForSeconds(d);
    if (cube2 != null) cube2.SetActive(false);
    if (cube3 != null) cube3.SetActive(false);

    yield return new WaitForSeconds(d);
    if (cube4 != null) cube4.SetActive(true);
    if (cube5 != null) cube5.SetActive(true);

    yield return new WaitForSeconds(d);
    if (cube6 != null) cube6.SetActive(true);
}

private void StartScoreWindow(float seconds)
{
    StopScoreWindow();
    scoreRoutine = StartCoroutine(ScoreWindowCoroutine(seconds));
}

private void StopScoreWindow()
{
    if (scoreRoutine != null)
    {
        StopCoroutine(scoreRoutine);
        scoreRoutine = null;
    }
}

private IEnumerator ScoreWindowCoroutine(float seconds)
{
    yield return new WaitForSeconds(Mathf.Max(0f, seconds));

    if (scoreStage == ScoreStage.WaitingForButton3)
    {
        currentScore = 0;
        UpdateScoreUI();
        scoreStage = ScoreStage.None;
    }
    else if (scoreStage == ScoreStage.WaitingForButton2)
    {
        UpdateScoreUI();
        scoreStage = ScoreStage.None;
    }

    scoreRoutine = null;
}

private void PlayClip(AudioClip clip)
```

```
{
    if (clip == null) return;

    if (audioSource == null)
    {
        AudioSource.PlayClipAtPoint(clip, transform.position);
        return;
    }

    audioSource.PlayOneShot(clip);
}

private void OnAnswerButtonPressed(bool pressedButton2, bool pressedButton3)
{
    if (scoreStage == ScoreStage.WaitingForButton3)
    {
        StopScoreWindow();

        if (pressedButton3)
        {
            currentScore = 10;
            PlayClip(correctPhase1Clip);
        }
        else
        {
            currentScore = 0;
        }

        UpdateScoreUI();
        scoreStage = ScoreStage.None;
        return;
    }

    if (scoreStage == ScoreStage.WaitingForButton2)
    {
        StopScoreWindow();

        if (pressedButton2)
        {
            currentScore = Mathf.Min(20, currentScore + 10);
            PlayClip(correctPhase2Clip);
        }

        UpdateScoreUI();
        scoreStage = ScoreStage.None;
        return;
    }
```

```
    }
}
```