Gamification of Computer Based Assessments: Variant 3

Appendix

Author: Joel Jolly

Course: BSc Computer Science

Supervisor: Dr. Senir Dinar

Student ID: 21067140

April 2024

Contents

Source Code

Source code along with all the assets used can be found using the link. It leads to a Google Drive folder which has the unity base file for the game, including all scenes, assets, animations and so on. In the submission session the game's executable version can be found:

https://drive.google.com/file/d/1Zfavs7j-2_qPCq0GsCa1Ig2-lR1vq1rj/view?usp=sharing

The link below leads to a Google Drive folder which has the executable version for the game:

https://drive.google.com/file/d/1xEdZLKbCwykHihazEL6hyyTQy0Uq0C9C/view?usp=sharing

The following is a source code for all the scripts:

1.1 GameFlow.cs

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

public class GameFlow : MonoBehaviour
{
    // Public Variables
    public Transform tile1Obj;
    public Transform spikesObj;
    public Transform statueObj;
    public Transform boulderObj;
    public Transform gateObj;
    public QuestionHandler questionHandler;
    public PlayerRun playerRun;
    public PlayerStatisitcs playerStatisitcs;
    public TextMeshProUGUI popupText;

    //Private Variables
    private Vector3 nextTileSpawn;
    private Vector3 nextSpikeSpawn;
    private Vector3 nextStatueSpawn;
    private Vector3 nextBoulderSpawn;
    private Vector3 nextGateSpawn;
    private float playerZPosition;
    private int boulderSpawnCounter = 0;
    private int maxBoulderSpawnInterval = 3;
    private bool hasChangedQuestion = false;

    void Start()
    {
        //Sets the intial values for the tile spawns
        nextTileSpawn.z = 24;
        nextTileSpawn.y = 0.9f;
        nextTileSpawn.x = 0.5f;
        playerZPosition = 5;
        //Starts spawning in the tiles that are not in the player's sight and starts timer to destroy premade
tiles
        StartCoroutine(SpawnTiles());
```

```
    StartCoroutine(DestroyPremadeTiles());
}

void Update()
{
    //Used to update the player's z position and updates time
    playerZPosition += 3f * Time.deltaTime;
    playerStatisitcs.UpdateScore(DifficultyChanger());
    //Checks if a gate collision has occured
    CheckIfPlayerAtGate();
}

void CheckIfPlayerAtGate()
    /**
        This function is used to check if the player is near the gate
    **/
{

    if (questionHandler == null)
    {
        Debug.LogError("QuestionHandler is not assigned to the GameFlow script.");
        return;
    }

    Collider[] colliders = Physics.OverlapSphere(new Vector3(0, 0, playerZPosition - 5), 0);

    //Checks for each of the gate and see if player has got to it
    foreach (Collider collider in colliders)
    {
        if (collider.CompareTag("Gate") && !hasChangedQuestion)
        {
            if (questionHandler != null)
            {
                //Requests for question and changes the question once the player has passed it.
                QuestionChecker(questionHandler.ReturnQuestionIndex());
                questionHandler.ChangeQuestion();
                hasChangedQuestion = true;
                StartCoroutine(ResetQuestionFlagAfterDelay());
            }
            else
            {
                Debug.LogError("QuestionHandler is null. Make sure it is assigned in the Unity Editor.");
            }
            break;
        }
    }
}

void QuestionChecker(int questionIndex)

{

    playerStatisitcs.IncrementQuestions();
    int playerXPosition = playerRun.GetPlayerPosition();
    int questionXPosition = questionIndex - 2;
    //Checks if the players X position and the questions X position matches
    if (playerXPosition == questionXPosition)
    {
        //Functions are called if the player gets the question right
        playerStatisitcs.IncrementAnsweredCorrectly();
        playerStatisitcs.UpdateScore(100);
        playerStatisitcs.UpdateQuestionList(questionHandler.GetCurrentQuestion() + "Correct");
        ShowPopUp("CORRECT", Color.green);
    }
    else
    {
```

```
        //Functions are called if the player gets the question wrong
        playerStatisitcs.UpdateQuestionList(questionHandler.GetCurrentQuestion() + "Incorrect");
        playerRun.ReduceLife();
        ShowPopUp("INCORRECT", Color.red);
    }
}

IEnumerator ResetQuestionFlagAfterDelay()
    /**
    This function is used as a delay, so questions are not constantly changing when approaching gate.
    **/
{
    yield return new WaitForSeconds(5f);
    hasChangedQuestion = false;
}

IEnumerator DestroyPremadeTiles()
    /**
    This function is used to destroy the tiles that are intially made in the game.
    **/
{
    // Tiles are destroyed after 10 seconds, usually the amount of time it takes for the player to move
    yield return new WaitForSeconds(10);

    Transform[] allObjects = GameObject.FindObjectsOfType<Transform>();
    foreach (Transform obj in allObjects)
    {
        if (obj.name.Contains("originaltile"))
        {
            Destroy(obj.gameObject);
        }
    }
}

IEnumerator SpawnTiles()
    /**
    This function is used to spawn the tiles and obstacles dynamically.
    **/
{
    float spawnDistance = 24f;

    while (true)
    {
        //Checks if the player is close to the last spawned tile
        if (playerZPosition + spawnDistance >= nextTileSpawn.z)
        {
            for (int i = 0; i < 6; i++)
            {
                //Used to spawn all the obstacles and tiles in regular intervals
                SpawnTile(tile1Obj, nextTileSpawn);
                SpawnTile(tile1Obj, nextTileSpawn);
                SpawnTile(tile1Obj, nextTileSpawn);
                SpawnTile(tile1Obj, nextTileSpawn);


                if (boulderSpawnCounter == 0)
                {
                    nextBoulderSpawn.z = nextTileSpawn.z;
                    nextBoulderSpawn.x = Random.Range(-1, 3) - 0.7f;
                    SpawnBoulder(boulderObj, nextBoulderSpawn);
                }
                boulderSpawnCounter = (boulderSpawnCounter + 1) % maxBoulderSpawnInterval;
                if (i % 3 == 1)
                {
                    nextGateSpawn.z = nextTileSpawn.z;
                    SpawnGate(gateObj, nextGateSpawn, -1);
```

```
                    SpawnGate(gateObj, nextGateSpawn, 0);
                    SpawnGate(gateObj, nextGateSpawn, 1);
                    SpawnGate(gateObj, nextGateSpawn, 2);
                }


                if (i % 3 != 1)
                {
                    SpawnObstacle(nextTileSpawn, statueObj, 3);
                    SpawnObstacle(nextTileSpawn, spikesObj, 3);
                }
            }
        }

        yield return null;
    }
}

int DifficultyChanger()
    /**
        This function is used to update the score difficulty based on the questions answered correctly.
    **/
{
    int totalQuestions = playerStatisitcs.GetQuestionAsked();
    if (totalQuestions == 0)
    {
        return 3;
    }

    //Students managed to have a multiplier of 5 points if they have an accuracy of 80% or more
    float correctPercentage = (playerStatisitcs.GetAnsweredCorrectly() / totalQuestions) * 100f;
    if (correctPercentage >= 80f)
    {
        playerStatisitcs.changeDifficulty("Hard");
        return 5;
    }

    //Students managed to have a multiplier of 3 points if they have an accuracy of 79-40%
    else if (correctPercentage >= 40f)
    {
        playerStatisitcs.changeDifficulty("Medium");
        return 3;
    }

    //Students managed to have a multiplier of 1 points if they have an accuracy of less than 39%
    else
    {
        playerStatisitcs.changeDifficulty("Easy");
        return 1;
    }
}

IEnumerator DestroyItem(GameObject obj)
    /**
        This function is used to destroy a desired gameObject
        Arg: obj- the game object they wish to destroy
    **/
{
    float deletionDistance = 10f;
    while (obj != null && (obj.transform.position.z + deletionDistance) > playerZPosition)
    {
        yield return null;
    }

    if (obj != null)
    {
```

```
        Destroy(obj);
    }
}

void SpawnTile(Transform obj, Vector3 spawnPosition)
    /**
        This function is used to spawn a tile on a field
        Args: obj - the tile object they wish to spawn
              spawnPosition - the position at which the object will spawn
    **/
{

    nextTileSpawn.z += 3;
    nextTileSpawn.y = 0.9f;
    nextTileSpawn.x = 0.5f;
    Transform tile = Instantiate(obj, spawnPosition, obj.rotation);
    StartCoroutine(DestroyItem(tile.gameObject));
}

void SpawnObstacle(Vector3 spawnPosition, Transform obstaclePrefab, int obstacleCount)
    /**
        This function is used to spawn an obstacle on a field
        Args: obstaclePrefab - the type of obstacle they want to spawn
              spawnPosition - the position at which the object will spawn
              obstacleCount - the number of obstacles they want to spawn
    **/
{
    float offsetZ = -3f;

    for (int i = 0; i < obstacleCount; i++)
    {
        //Used to spawn objects dynamically across the lanes in random z axis
        Vector3 obstacleSpawnPosition = spawnPosition + new Vector3(Random.Range(-1, 3), 0, offsetZ * i);

        if (obstacleSpawnPosition.z == nextStatueSpawn.z || obstacleSpawnPosition.z == nextSpikeSpawn.z)
        {
            int shift = Random.Range(-1, 1);
            if (shift == 0)
            {
                obstacleSpawnPosition.z += 1;
            }
            else
            {
                obstacleSpawnPosition.z -= 1;
            }
        }
        //Spawns the correct type of obstacle determined by the input
        if (obstaclePrefab == statueObj)
        {
            nextStatueSpawn = obstacleSpawnPosition;
            SpawnStatue(obstaclePrefab, nextStatueSpawn);
        }
        else if (obstaclePrefab == spikesObj)
        {
            nextSpikeSpawn = obstacleSpawnPosition;
            SpawnSpikes(obstaclePrefab, nextSpikeSpawn);
        }
    }
}

void SpawnSpikes(Transform obj, Vector3 spawnPosition)
    /**
        This function is used to spawn a spike on a field
        Args: obj - the tile object they wish to spawn
              spawnPosition - the position at which the object will spawn
    **/
```

```
{
    nextSpikeSpawn = spawnPosition;
    nextSpikeSpawn.y = 0;
    nextSpikeSpawn.x = spawnPosition.x - 0.5f;
    Transform spikes = Instantiate(obj, nextSpikeSpawn, obj.rotation);
    StartCoroutine(DestroyItem(spikes.gameObject));
}

void SpawnStatue(Transform obj, Vector3 spawnPosition)
    /**
        This function is used to spawn a statue on a field
        Args: obj - the tile object they wish to spawn
              spawnPosition - the position at which the object will spawn
    **/
{
    nextStatueSpawn = spawnPosition;
    nextStatueSpawn.y = 0;
    nextStatueSpawn.x = spawnPosition.x - 0.5f;
    Transform statue = Instantiate(obj, nextStatueSpawn, obj.rotation);
    StartCoroutine(DestroyItem(statue.gameObject));
}

void SpawnBoulder(Transform obj, Vector3 spawnPosition)
{
    /**
        This function is used to spawn a boulder on a field
        Args: obj - the tile object they wish to spawn
              spawnPosition - the position at which the object will spawn
    **/

    Transform boulder = Instantiate(obj, spawnPosition, obj.rotation);
    StartCoroutine(DestroyItem(boulder.gameObject));

    Collider boulderCollider = boulder.GetComponent<Collider>();
    if (boulderCollider != null)
    {
        boulderCollider.isTrigger = true;
        boulder.gameObject.AddComponent<BoulderCollisionHandler>();
    }
}

void SpawnGate(Transform obj, Vector3 spawnPosition, int gateNumber)
{
    /**
        This function is used to spawn a gate on a field
        Args: obj - the tile object they wish to spawn
              spawnPosition - the position at which the object will spawn
              gateNumber - used to identify which gate (1 of 4) is used to spawn on the environment
    **/
    nextGateSpawn = spawnPosition;
    nextGateSpawn.x = gateNumber;
    nextGateSpawn.y = -0.05f;
    Transform gate = Instantiate(obj, nextGateSpawn, obj.rotation);
    StartCoroutine(DestroyItem(gate.gameObject));
}

IEnumerator HidePopUpAfterDelay(float delay)
    /**
        This function is used to hide the pop up message after it has been loaded up
        Args: delay - how long the message should wait till popping up
    **/
{
    yield return new WaitForSeconds(delay);
    popupText.gameObject.SetActive(false);
}
```

```
    void ShowPopUp(string message, Color color)
        /**
            This function is used to make a pop up to show if the student got it right or wrong
            Args: message - the message that shows if the student got the question right or wrong
                  color - the color of the message
        **/
    {

        popupText.text = message;
        popupText.color = color;
        popupText.gameObject.SetActive(true);
        StartCoroutine(HidePopUpAfterDelay(2f));
    }
}
```

1.2 JsonFileLoader.cs

```
using UnityEngine;
using UnityEngine.Events;
using System;
using System.IO;
using System.Collections.Generic;
using TMPro;
using SFB;
using Newtonsoft.Json;

public class JsonFileLoader : MonoBehaviour
{
    //Text objects used for displaying the options and questions
    public TextMeshProUGUI questionText;
    public TextMeshProUGUI option1;
    public TextMeshProUGUI option2;
    public TextMeshProUGUI option3;
    public TextMeshProUGUI option4;

    public string answer;
    private List<Question> questions;
    private int currentQuestionIndex = 0;
    private string locationFilePath = Path.Combine(Application.streamingAssetsPath, "location.json");
    private string jsonFilePath = "";
    public class JsonLoadedEvent : UnityEvent<List<Question>> { }

    void Start()
    {
        resetQuestion();
    }

    public void LoadJsonFile()
         /**
            This function is used to load the JSON file
        **/
    {
        //Opens up a file dialog to select JSON file
        string[] paths = StandaloneFileBrowser.OpenFilePanel("Open JSON File", "", "json", false);

        //Does error checking to see if the file is appropriate
        if (paths != null && paths.Length > 0)
        {
            string locationJson = paths[0];

            if (!string.IsNullOrEmpty(locationJson))
            {
                try
                {
                    string jsonContent = File.ReadAllText(locationJson);


                    //Attempt to deserialize the JSON content
                    try
                    {
                        List<Question> loadedQuestions =
JsonConvert.DeserializeObject<List<Question>>(jsonContent);

                        //Checks if the loaded questions are in the correct format
                        if (loadedQuestions != null && loadedQuestions.Count > 0 &&
loadedQuestions[0].IsValid())
                        {
                            UpdateQuestions(loadedQuestions);
                            jsonFilePath = locationJson;
```

```
                    }
                    else
                    {
                        Debug.LogError("Invalid questions format. Questions remain unchanged.");
                    }
                }
                catch (JsonSerializationException)
                {
                    Debug.LogError("Error loading JSON file: Invalid JSON format.");
                }
            }
            catch (Exception e)
            {
                Debug.LogError($"Error loading JSON file: {e.Message}");
            }
        }
    }
}

private void UpdateQuestions(List<Question> updatedQuestions)
    /**
        This function is used to update the question list
        Args: updateQuestions: contains a list of questions that needs to updated
    **/
{
    if (updatedQuestions != null && updatedQuestions.Count > 0)
    {
        questions = updatedQuestions;
        currentQuestionIndex = 0;
        DisplayCurrentQuestion();
    }
    else
    {
        Debug.LogError("Invalid questions list or list is empty. Cannot update questions.");
    }
}

public void SaveJsonFile()
    /**
        This function is used to save the json file that's been created
    **/
{
    try
    {
        if (questions != null && questions.Count > 0)
        {
            //Used to serialize the current jsonFilePath to a JSON object
            var locationData = new { filePath = jsonFilePath };
            string locationJson = JsonConvert.SerializeObject(locationData);
            File.WriteAllText(locationFilePath, locationJson);
        }
        else
        {
            Debug.LogError("No questions loaded.");
        }
    }
    catch (Exception e)
    {
        Debug.LogError($"Error saving JSON file location: {e.Message}");
    }
}

void LoadQuestionsFromJson()
    /**
        This function is used load the questions from the JSON file
```

```
        **/
    {
        //Error checking to see if file exists, or any potential errors that occured while loading JSON file
        try
        {
            if (File.Exists(locationFilePath))
            {

                string locationJson = File.ReadAllText(locationFilePath);
                LocationData locationData = JsonConvert.DeserializeObject<LocationData>(locationJson);
                jsonFilePath = locationData.filePath;
                if (jsonFilePath == "question.json")
                {
                    jsonFilePath = Path.Combine(Application.streamingAssetsPath, "question.json");
                }

                //Checks if the JSON file path is valid
                if (File.Exists(jsonFilePath))
                {
                    //Used to deserialize JSON file into a list of Question objects
                    string jsonContent = File.ReadAllText(jsonFilePath);
                    questions = JsonConvert.DeserializeObject<List<Question>>(jsonContent);

                    if (questions == null || questions.Count == 0)
                    {
                        Debug.LogError("No questions found or questions list is empty.");
                        return;
                    }
                }
                else
                {
                    Debug.LogError("Specified JSON file does not exist.");
                    return;
                }
            }
            else
            {
                Debug.LogError("Location file not found.");
                return;
            }
        }
        catch (Exception e)
        {
            Debug.LogError($"An error occurred while loading questions: {e.Message}");
            return;
        }
    }

    public class LocationData
    {
        /**
            This class is used to store the filePath

        **/
        public string filePath { get; set; }
    }

    void DisplayCurrentQuestion()
        /**
            This function is used to display the current questions
        **/
    {
        //Checks if the questions list is initialized and the current index is within bounds
        if (questions != null && questions.Count > 0 && currentQuestionIndex >= 0 && currentQuestionIndex <
questions.Count)
        {
```

```
        Question currentQuestion = questions[currentQuestionIndex];

        questionText.text = currentQuestion.question;
        answer = currentQuestion.correctAnswer;
        SetOptionText(option1, currentQuestion.options[0], answer);
        SetOptionText(option2, currentQuestion.options[1], answer);
        SetOptionText(option3, currentQuestion.options[2], answer);
        SetOptionText(option4, currentQuestion.options[3], answer);
    }
    else
    {
        Debug.Log("Invalid index or questions list is not properly initialized.");
    }
}

void MoveToNextQuestion()
    /**
        This function is used to move to the next question
    **/
{
    currentQuestionIndex++;

    if (currentQuestionIndex >= questions.Count)
    {
        currentQuestionIndex = 0;
    }

    DisplayCurrentQuestion();
}
public void OnNextQuestionButtonClick()
{
    MoveToNextQuestion();
}

void MoveToBackQuestion()
    /**
        This function is used to move to the previous question
    **/
{
    currentQuestionIndex--;
    if (currentQuestionIndex < 0)
    {
        currentQuestionIndex = questions.Count - 1;
    }
    DisplayCurrentQuestion();

}

public void OnBackQuestionButtonClick()
{
    MoveToBackQuestion();
}

void SetOptionText(TextMeshProUGUI optionText, string option, string answer)
/**
        This function is used to set the color for the text, green if it is correct.
        Args: optionText - the textUI that is used to be diplayed
              option - the string for the option
              answer - the correct answer for the question
    **/
{
    if (optionText != null)
    {
        optionText.text = option;
        if (option == answer)
        {
```

```
                optionText.color = Color.green;
            }
            else
            {
                optionText.color = Color.white;
            }
        }
    }

    public void resetQuestion()
    /**
            This function is used to reset the question information when reloading the question list page
        **/
    {
        locationFilePath = Path.Combine(Application.streamingAssetsPath, "location.json");;
        jsonFilePath = "";
        LoadQuestionsFromJson();
        DisplayCurrentQuestion();
    }
}
```

1.3 MainMenu.cs

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

public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        SceneManager.LoadScene("Game");
    }

    public void QuitGame()
    {
        Application.Quit();
    }

}
```

1.4 PlayerRun.cs

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

public class PlayerRun : MonoBehaviour
{

    private bool isMoving = false;
    private static int livesCount;
    private int playerPosition;

    public TextMeshProUGUI playerLives;
    public TextMeshProUGUI playerScore;
    public PlayerStatisitcs playerStatisitcs;

    void Start()
    {
        playerPosition = 0;
        livesCount = 10;
        playerLives.text = "Lives: " + livesCount.ToString();
        playerScore.text = "Score: 0";

        MoveForward();
    }

    void Update()
    {
        if (!isMoving)
        {

            //Uses input from left arrow key to move player left
            if (Input.GetKey(KeyCode.LeftArrow) && playerPosition != -1)
            {
                StartCoroutine(MoveSideways(-1));
            }
            //Uses input from right arrow key to move player right
            if (Input.GetKey(KeyCode.RightArrow) && playerPosition != 2)
            {
                StartCoroutine(MoveSideways(1));
            }
            //Uses input from space key to make player jump
            if (Input.GetKeyDown("space"))
            {
                GetComponent<Rigidbody>().velocity = new Vector3(0, 1, 3);
                StartCoroutine(stopJump());
            }
        }

        //Reset game if lives are depleted
        if (livesCount <= 0)
        {
            ResetGame();
        }

        playerScore.text = "Score: " + playerStatisitcs.GetScore().ToString();
    }

    // Get current player position
    public int GetPlayerPosition()
        /**
```

```
        This function is used return the player position
    **/
{
    return playerPosition;
}

IEnumerator MoveSideways(int direction)
    /**
        This function is used to move the player sideways
        Args: direction- a value to indicate which side the play should move
    **/
{
    isMoving = true;

    Vector3 targetVelocity = new Vector3(direction, 0, 3);
    GetComponent<Rigidbody>().velocity = targetVelocity;

    yield return new WaitForSeconds(1);

    GetComponent<Rigidbody>().velocity = new Vector3(0, 0, 3);
    isMoving = false;

    playerPosition += direction;
}

IEnumerator stopJump()
    /**
        This function is used to make the player jump and then come back down.
    **/
{
    isMoving = true;
    yield return new WaitForSeconds(0.75f);
    GetComponent<Rigidbody>().velocity = new Vector3(0,-1,3);
    yield return new WaitForSeconds(0.75f);
    GetComponent<Rigidbody>().velocity = new Vector3(0,0,3);
    isMoving = false;
}

void MoveForward()
    /**
        This function is used to move the player forward
    **/
{
    GetComponent<Rigidbody>().velocity = new Vector3(0, 0, 3);
}

void OnTriggerEnter(Collider other)
    /**
        This function is used to check there has been a collision, and if so reduce a life.
    **/
{
    if (other.CompareTag("Obstacle"))
    {
        ReduceLife();
        Destroy(other.gameObject);
    }
}

public void ReduceLife()
    /**
        This function is used reduce the number of live a player has ingame.
    **/
{
    livesCount --;
    playerLives.text = "Lives: " + livesCount.ToString();
}
```

```
    void ResetGame()
        /**
            This function is used reset the player stats and sends the player back to the game menu.
        **/
    {
        // Reset player stats
        playerStatisitcs.ResetStats();
        playerPosition = 0;
        transform.position = new Vector3(0, 0, 0);

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

1.5 PlayerStatistics.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using Newtonsoft.Json;

public class PlayerStatisitcs : MonoBehaviour
{

    private int answeredCorrectly;
    private int questionsAsked;
    private string questionDifficulty;
    private int scoreCount;
    private string questionList;
    private static bool saved;

    void Start()
    {
        answeredCorrectly = 0;
        questionsAsked = 0;
        questionDifficulty = "Medium";
        scoreCount = 0;
        questionList = "";
        saved = true;
    }

    public int GetScore()
        /**
            This function is used return the player score
        **/
    {
        return scoreCount;
    }

    public int GetQuestionAsked()
        /**
            This function is used return the number of questions asked
        **/
    {
        return questionsAsked;
    }

    public int GetAnsweredCorrectly()
        /**
            This function is used return the number of quesitions answered correctly
        **/
    {
        return answeredCorrectly;
    }

    public void changeDifficulty(string difficulty)
        /**
            This function is used change the score difficulty of the current run.
            Args: difficulty - the difficulty it changes to
        **/
    {
        questionDifficulty = difficulty;
    }

    public void UpdateQuestionList(string question)
        /**
            This function is used to add questions to the question list
```

```
        Args: question - the question that needs to be added
    **/
{
    questionList = questionList + ";" + question;
}

public void UpdateScore(int updateValue)
    /**
        This function is used update the score of the run
        Args: updateValue - the number that the score must be added to
    **/
{
    scoreCount += updateValue;
}

public void IncrementAnsweredCorrectly()
    /**
        This function is used increment the number of questions asked correctly
    **/
{
    answeredCorrectly++;
}

public void IncrementQuestions()
    /**
        This function is used to increment the number of questions asked
    **/
{
    questionsAsked++;
}

public void ResetStats()
    /**
        This function is used to reset the statistics of the run
    **/
{
    if (saved)
    {
        UpdateRunToJson();
    }
    scoreCount = 0;
    questionsAsked = 0;
    answeredCorrectly = 0;
}

void UpdateRunToJson()
    /**
        This function is used to update run statistics into the Json file
    **/
{
    saved = false;
    string filePath = Path.Combine(Application.streamingAssetsPath, "runs.json");;
    List<GameData> gameDataList;

    if (File.Exists(filePath))
    {
        string jsonData = File.ReadAllText(filePath);

        if (!string.IsNullOrEmpty(jsonData))
        {

            gameDataList = JsonConvert.DeserializeObject<List<GameData>>(jsonData);
        }
        else
        {
            gameDataList = new List<GameData>();
```

```
            }
        }
        else
        {
            gameDataList = new List<GameData>();
        }

        //Creates an instance of GameData and saves the new statistics
        GameData newGameData = new GameData();
        newGameData.score = scoreCount;
        newGameData.questionsAsked = questionsAsked;
        newGameData.questionsCorrect = answeredCorrectly;
        newGameData.questionDifficulty = questionDifficulty;
        newGameData.questionList = questionList;

        gameDataList.Add(newGameData);
        string updatedJsonData = JsonConvert.SerializeObject(gameDataList, Formatting.Indented);
        File.WriteAllText(filePath, updatedJsonData);
    }
}

public class GameData
    /**
        This class is used to represent game data for Json serialization purposes
    **/
{
    public float score;
    public int questionsAsked;
    public int questionsCorrect;
    public string questionDifficulty;
    public string questionList;
}
```

1.6 Question.cs

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

public class Question
{
    public string question { get; set; }
    public List<string> options { get; set; }
    public string correctAnswer { get; set; }

    public override string ToString()
    {
        return $"{question}\nOptions: {string.Join(", ", options)}\nCorrect Answer: {correctAnswer}\n";
    }

    public bool IsValid()
        /**
            This function is used to check if the question is valid
        **/
    {
        if (string.IsNullOrEmpty(question) || options == null || options.Count != 4 ||
string.IsNullOrEmpty(correctAnswer))
        {
            return false;
        }
        foreach (string option in options)
        {
            if (string.IsNullOrEmpty(option))
            {
                return false;
            }
        }

        if (!options.Contains(correctAnswer))
        {
            return false;
        }

        return true;
    }


}
```

1.7 QuestionHandler.cs

```
using System;
using System.Collections.Generic;
using Newtonsoft.Json;
using TMPro;
using UnityEngine;
using System.IO;

public class QuestionHandler : MonoBehaviour
{
    public TextMeshProUGUI questionText;
    public TextMeshProUGUI option1;
    public TextMeshProUGUI option2;
    public TextMeshProUGUI option3;
    public TextMeshProUGUI option4;
    public string answer;
    private string jsonFilePath = "";
    private string locationFilePath = Path.Combine(Application.streamingAssetsPath, "location.json");
    private int currentQuestionIndex = 0;

    public List<Question> questions;

    void Start()
    {
        LoadQuestionsFromJson();
        DisplayCurrentQuestion();
    }

    void Update()
    {
    }

    void LoadQuestionsFromJson()
        /**
            This function is used to load questions from a Json file.
        **/

    {
        try
        {
            if (File.Exists(locationFilePath))
            {
                string locationJson = File.ReadAllText(locationFilePath);
                LocationData locationData = JsonConvert.DeserializeObject<LocationData>(locationJson);
                jsonFilePath = locationData.filePath;

                if (jsonFilePath == "question.json")
                {
                    jsonFilePath = Path.Combine(Application.streamingAssetsPath, "question.json");
                }

                if (File.Exists(jsonFilePath))
                {
                    string jsonContent = File.ReadAllText(jsonFilePath);
                    questions = JsonConvert.DeserializeObject<List<Question>>(jsonContent);

                    if (questions == null || questions.Count == 0)
                    {
                        Debug.LogError("No questions found or questions list is empty.");
                        return;
                    }
                }
                else
```

```
                {
                    Debug.LogError("Specified JSON file does not exist.");
                    return;
                }
            }
            else
            {
                Debug.LogError("Location file not found.");
                return;
            }
        }
        catch (Exception e)
        {
            Debug.LogError($"An error occurred while loading questions: {e.Message}");
            return;
        }
    }

    public class LocationData
        /**
            This class is used to represent the data structure of the location Json file
        **/
    {
        public string filePath { get; set; }
    }

    void DisplayCurrentQuestion()
        /**
            This function is used to display the current question and options
        **/
    {
        if (questions != null && questions.Count > 0 && currentQuestionIndex >= 0 && currentQuestionIndex
< questions.Count)
        {
            Question currentQuestion = questions[currentQuestionIndex];

            questionText.text = currentQuestion.question;
            answer = currentQuestion.correctAnswer;
            option1.text = currentQuestion.options[0];
            option2.text = currentQuestion.options[1];
            option3.text = currentQuestion.options[2];
            option4.text = currentQuestion.options[3];
        }
        else
        {
            Debug.Log("Invalid index or questions list is not properly initialized.");
        }
    }

    public int ReturnQuestionIndex()
        /**
            This function is used to return the index of the correct answer in the options array
        **/
    {
        Question currentQuestion = questions[currentQuestionIndex];
        int correctIndex = -1;
        for (int i = 0; i < 3; i++)
        {
            if (currentQuestion.options[i] == answer)
            {
                correctIndex = i + 1;
                break;
            }
        }
        return correctIndex;
    }
```

```
    public string GetCurrentQuestion()
        /**
            This function is used to get the current question
        **/
    {
        Question currentQuestion = questions[currentQuestionIndex];
        return currentQuestion.question;
    }

    public void ChangeQuestion()
        /**
            This function is used to change the current question
        **/
    {
        currentQuestionIndex = (currentQuestionIndex + 1) % questions.Count;
        DisplayCurrentQuestion();
    }
}
```

1.8 SettingMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Newtonsoft.Json;
using System.IO;

public class SettingsMenu : MonoBehaviour
{
    private string filePath;
    void Start()
    {

    }


    void Update()
    {

    }

    public void ResetRunsFile()
        /**
            This function is used to reset the runs json file and make it empty
        **/
    {
        filePath = "Assets/Resources/runs.json";
        List<GameData> gameDataList = new List<GameData>();
        string updatedJsonData = JsonConvert.SerializeObject(gameDataList, Formatting.Indented);
        File.WriteAllText(filePath, updatedJsonData);
    }

    public void ResetQuestionLocation()
        /**
            This function is used to reset the question location to the default location
        **/
    {
        filePath = "Assets/Resources/location.json";
        string newFilePath = "Assets\\Resources\\question.json";
        var jsonObject = new { filePath = newFilePath };
        string updatedJsonData = JsonConvert.SerializeObject(jsonObject);
        File.WriteAllText(filePath, updatedJsonData);
    }

}
```

1.9 TeacherReport.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using Newtonsoft.Json;
using TMPro;
using UnityEngine.UI;

public class TeacherReport : MonoBehaviour
{

    public TextMeshProUGUI runNumber;
    public TextMeshProUGUI questionsAsked;
    public TextMeshProUGUI answeredCorrectly;
    public TextMeshProUGUI questionDifficulty;
    public TextMeshProUGUI scoreCount;
    public TextMeshProUGUI improvementValue;
    public TextMeshProUGUI percentageValue;
    public TextMeshProUGUI questionAnswerText;

    private List<GameData> gameDataList;
    private int currentRunIndex = 0;

    void Start()
    {
        string filePath = Path.Combine(Application.streamingAssetsPath, "runs.json");;
        string jsonData = File.ReadAllText(filePath);
        gameDataList = JsonConvert.DeserializeObject<List<GameData>>(jsonData);
        DisplayCurrentRun();
    }

    void DisplayCurrentRun()
        /**
            This function is used to display the information about the current runs
        **/
    {
        if (gameDataList != null && gameDataList.Count > currentRunIndex)
        {
            GameData currentRun = gameDataList[currentRunIndex];

            //Displays run information
            runNumber.text = "Run #" + (currentRunIndex + 1);
            questionsAsked.text = "Questions Asked: " + currentRun.questionsAsked;
            answeredCorrectly.text = "Answered Correctly: " + currentRun.questionsCorrect;
            questionDifficulty.text = "Score Difficulty: " + currentRun.questionDifficulty;
            scoreCount.text = "Score: " + currentRun.score;

            //Calculates and display improvement percentage
            float improvement = CalculateImprovement(currentRunIndex);
            improvementValue.text = "Improvement: " + improvement.ToString("0.00") + "%";

            //Calculates and display percentage of correct answers
            float percentage = CalculatePercentage(currentRun);
            percentageValue.text = "Percentage Correct: " + percentage.ToString("0.00") + "%";

            //Parse and display individual questions and answers
            string questionText = ParseQuestionList(currentRun.questionList);
            questionAnswerText.text = questionText;
        }
    }

    // Method to parse question list string and format it for display
```

```
    string ParseQuestionList(string questionList)
        /**
            This function is used to parse a question list into question and correctness
            Args: questionList - the list of quesitons and correctness to be parsed
        **/
    {
        string parsedText = "";
        string[] questionEntries = questionList.Split(';');
        int questionNumber = 1;
        foreach (var entry in questionEntries)
        {
            string[] parts = entry.Split('?');
            if (parts.Length == 2)
            {
                string question = parts[0];
                string correctness = parts[1];
                parsedText += questionNumber + ". " + question.Trim() + " (" +
(correctness.Trim().Equals("Correct") ? "Correct" : "Incorrect") + ")\n";
                questionNumber++;
            }
        }
        return parsedText;
    }

    float CalculateImprovement(int index)
        /**
            This function is used to calculate the improvement of a run, based on the previous run
            Args: index - the index of the run to compare it to its previous run
        **/
    {
        if (index <= 0)
            return 0;

        float currentScore = gameDataList[index].score;
        float previousScore = gameDataList[index - 1].score;

        if (previousScore == 0)
            return 0;

        float percentageImprovement = ((currentScore - previousScore) / previousScore) * 100;

        return percentageImprovement;
    }

    float CalculatePercentage(GameData data)
    /**
        This function is used to calculate the percentage of question answered correctly
    **/
    {
        return (data.questionsCorrect / (float)data.questionsAsked) * 100f;
    }

    void MoveToNextRun()
        /**
            This function is used move onto the next run and displays its information
        **/
    {
        currentRunIndex++;
        if (currentRunIndex >= gameDataList.Count)
        {
            currentRunIndex = 0;
        }
        DisplayCurrentRun();
    }

    public void OnNextRunButtonClick()
```

```
    {
        MoveToNextRun();
    }

    void MoveToBackRun()
        /**
            This function is used to move into its previous run and displays its information
        **/
    {
        currentRunIndex--;
        if (currentRunIndex < 0)
        {
            currentRunIndex = gameDataList.Count - 1;
        }
        DisplayCurrentRun();
    }
    public void OnBackRunButtonClick()
    {
        MoveToBackRun();
    }
}
```

1.10 AnimCon.cs

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

public class Animation : MonoBehaviour
{
    void Start()
    {
        // Gets the Animator component attached to the GameObject this script is attached to and plays the
animation named "Run".
        GetComponent<Animator>().Play("Run");
    }


    void Update()
    {

    }
}
```

1.11 BoulderMovement.cs

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

public class BoulderMovement : MonoBehaviour
{
    Rigidbody boulder;

    void Start()
    {

        boulder = GetComponent<Rigidbody>();
    }

    void Update()
    {
        //Uses the Rigidbody of the boulder and moves it 2 steps in the z axis towards the player.
        boulder.velocity = new Vector3(0, 0, -2);
    }
}
```

1.12 BoulderCollisionHandler.cs

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

public class BoulderCollisionHandler : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        //Used to check if there's been a collision with a boulder
        if (other.CompareTag("Obstacle"))
        {
            Destroy(other.gameObject); //Destroy obstalces with "Obstacle" tag that gets in the way of
boulder
        }
    }
}
```

User Manual:

This is a user manual of what to do when opening the game. Due to there being two user, students, and teacher, I will explain what each user would have to do.

2.1 Teacher User Guide

1. After unzipping the folder, open the Endless Runner.exe file which will load the user up into the game

2. Once in the game it is recommended that the teacher goes into Settings and recommended that the teacher clears both the runs and questions.

3. They can go back to the main menu screen from where they can go into the Teacher section and click on Question Bank

4. Once on the Question Bank page they can click on Import Json File navigate to a JSON that matches the serialization of the checker. If an example question file is wanted, you can go to the EndlessRunner_Data > StreamingAssets > question2.json file which is an example set of questions.

5. Once the question has been loaded, you can either navigate through the question base, or save the file, by clicking Save Json File.

6. Once the following steps have been completed, the teacher can hand over the game to the student who can play the game as they wish.

7. After the students have completed a couple runs, the teacher can now open the Student Analysis section of the Teacher page, where they can see the runs the student has made.

8. They will see a set of statistics for each run, which using the side buttons they can shuffle through.

9. By pressing the see questions button, they teacher can also see the questions answers along with its correctness.

2.2 Student User Guide

1. For the student, once the teacher has set everything up, they can press the play button which will result them being taken into the game itself.

2. Once loaded in they can use the left and right arrow keys to move the character and avoid the obstacles

3. They must avoid till they reach the gates. Once the gates have been reached, they stay in the correct lane of which they think the answer is. The question will be loaded as soon as the game starts and have more than enough time to read and pick an answer.

4. They play the game till, all their lives have been used up, then they will returned back to the main menu.