



## **Trivia Game Project**

# **Realisation**

Prepared by  
**Simeon Markov**  
Date: 2025-09-27

## I. Introduction

This document outlines the product development plan for **Trivia Game**, detailing the technical structure, implementation strategy, and development workflow. It includes the project's structure, code snippets displaying vital functionalities, tools, techniques.

## II. Project structure

Core components:

- **'NativeTriviaController.cs'** - Main game logic (UI Toolkit version).
- **'GameRulesController.cs'** – *Initializing game rules elements, main logic for content and navigation.*
- **'TriviaManager.cs'** - Bridge system & WebView integration manager.
- **'TriviaUIController.cs'** - Traditional Unity UI overlay controller.
- **'TriviaQuestionData.cs'** - Question database management system.
- **'TriviaGameUI.uxml'** - UI Toolkit layout file.
- **'TriviaGameStyles.uss'** - UI Toolkit styles (Converted from Tailwind).

System Architecture:

Assets/

```
|— Adaptive Performance/
|   |— Settings/
|       |— Simulator Provider Settings.asset
|— DefaultVolumeProfile.asset
|— InputSystem_Actions.inputactions
|— Plugins/
|   |— WebGL/
|       |— webgl-webview.jslib
|— Resources/
|— Scenes/
|   |— SampleScene.unity
|— Scripts/
|   |— Tests/
|       |— TriviaManagerTests.cs
|   |— Trivia/
|       |— DeepLinkManager.cs
|       |— GameRulesController.cs
|       |— GameSessionData.cs
|       |— JsonHelper.cs
```

- NativeTriviaController.cs
  - OpenTDBData.cs
  - TriviaLauncherExample.cs
  - TriviaManager.cs
  - TriviaQuestionData.cs
  - TriviaQuestionShared.cs
  - TriviaSystemInstaller.cs
  - TriviaUIController.cs
- Settings/
  - Lit2DSceneTemplate.scenetemplate
  - Renderer2D.asset
  - Scenes/
    - URP2DSceneTemplate.unity
  - UniversalRP.asset
- StreamingAssets/
  - trivia-questions.json
- UI/
  - USS/
    - GameRulesStyles.uss
    - TriviaGameStyles.uss
  - UXML/
    - TriviaGameUI.uxml
- UI Toolkit/
  - PanelSettings.asset
  - UnityThemes/
    - UnityDefaultRuntimeTheme.tss
- Unity.VisualScripting.Generated/
  - VisualScripting.Core/
    - AotStubs.cs
  - VisualScripting.Flow/
    - UnitOptions.db
- UniversalRenderPipelineGlobalSettings.asset
- \_Recovery/
  - 0.unity

### III. Technologies Used

#### A. Programming languages

The backend is solely on C# and for the UI were used the Unity UI Toolkit and custom CSS for styling.

#### B. Development platform

For development environment is used **.Net 8.0 Runtime (Long-Term Support)**

## C. Tools

### Unity Version Control:

The project uses the Unity Version Control system to keep track of project's history, changes, and contributions.

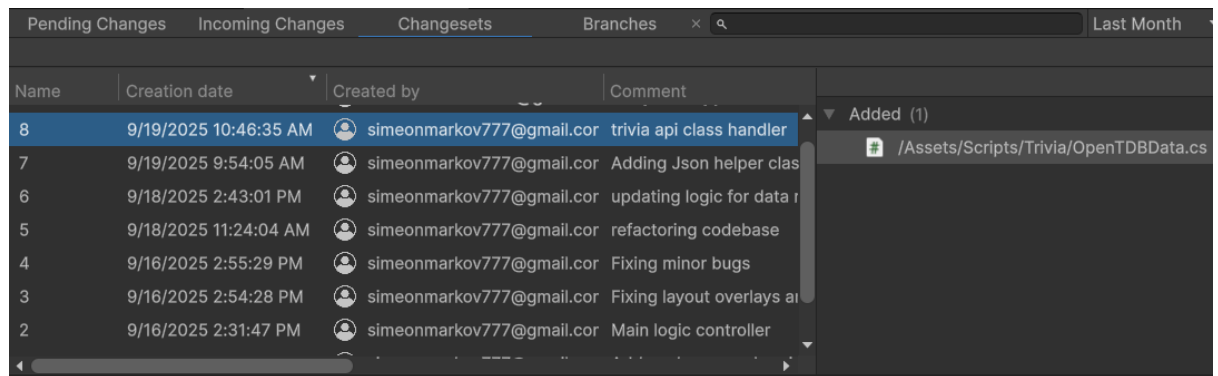


Figure 1: Snippet from Unity Version Control dialog

In figure 1 is shown the module dialog of commits (Changesets) where in a table are displayed the date of the commit, the contributor, the message (comment) and affected file/s. There are different tabs for showing pending changes (commits) and different branches.

## D. IDEs

**Visual Studio:** Used for writing C# code, which Unity is powered by. It is a suitable choice due to its broad features like debugging, compiling, testing, managing version control, etc. in one place. Moreover, it supports cross-platform development.

**Unity:** Used for creating native games for cross-platforms. It features a rich assets store and integration with third-party tools and plugins.

## IV. Design vs. Realization Differences

The initial game design, illustrated by the wireframes did take changes. The significant change was the way the game rules were being presented. In the first design the rules were appearing alongside the game menu, and now in the new improved design, there is a button, which when clicked, navigates the player to a dedicated window with different tabs containing information for the elements of the game like scoring, settings and tips and strategies.

## V. Code documentation

### A. Game Elements / Settings

For declaration of the game's elements is used Unity UI elements module.

```
// Game elements
private UnityEngine.UIElements.Label scoreText;
private ProgressBar progressBar;
private UnityEngine.UIElements.Label questionNumber;
private UnityEngine.UIElements.Label categoryBadge;
private UnityEngine.UIElements.Label difficultyBadge;
private UnityEngine.UIElements.Label questionText;
private UnityEngine.UIElements.Button[] answerButtons;
private VisualElement answerFeedback;
private UnityEngine.UIElements.Label feedbackEmoji;
private UnityEngine.UIElements.Label feedbackTitle;
private VisualElement questionNavigation;
private UnityEngine.UIElements.Button nextQuestionButton;
```

Figure 2: Code snippet of the game elements declaration.

Explanation: The fields are private (visible only within the class scope). The reason behind this is to keep encapsulation of data.

API settings: In this section are initialised the API url for the questions and categories.

```
[Header("API Settings")]
public bool useAPI = true;
public string apiUrl = "https://opentdb.com/api.php?amount=50&type=multiple&encode=url3986";
public string categoriesUrl = "https://opentdb.com/api_category.php";
private bool isLoading = false;
```

Figure 3: Snippet of API configuration

Explanation: Defined are the API url (Open Trivia DB API), which is set to fetch 50 questions per call (max fetching per call according to the API documentation: [https://opentdb.com/api\\_config.php](https://opentdb.com/api_config.php)), type (set to multi-choice) and encoding. Moreover, all possible categories are also fetched from the API, loading time for fetching the questions is initialised. The available categories are stored under dictionary data structure, because of the advantage of storing data in key-value pair.

Game settings: In this section are presented the questions, shuffling questions.

```
[Header("Game Settings")]
public TriviaQuestion[] triviaQuestions;
public int questionsPerGame = 10;
public bool shuffleQuestions = true;
```

*Figure 4: Game settings*

Explanation: The questions are stored in an array, the number of questions per game is set to a certain value (10) and a flag variable is initialised for helping in checking the state of shuffling.

## **B. Main logic**

In this section are presented the core functionalities of the project like fetching, querying questions, categories and shuffling method. Furthermore, scoring system is also implemented with methods for updating and saving the players progress as well as there is a method for escaping html attributes from the API for raw data.

## Method **SetupGameWithLoadedQuestions ()**:

```
List<TriviaQuestion> filteredQuestions;

if (string.IsNullOrEmpty(selectedCategory) || selectedCategory == "All Categories")
{
    // Use all questions if no category selected
    filteredQuestions = triviaQuestions.ToList();
}
else
{
    // Filter by selected category (case-insensitive comparison)
    filteredQuestions = triviaQuestions
        .Where(q => q.category.Equals(selectedCategory, StringComparison.OrdinalIgnoreCase))
        .ToList();

    // If no questions found in selected category, fall back to all questions
    if (filteredQuestions.Count == 0)
    {
        filteredQuestions = triviaQuestions.ToList();
    }
}
```

Figure 5: First part of the SetupGameWithLoadedQuestions()

Explanation: A list of questions is initialised for storing the filtered version (copy) of the fetched questions. Then the method checks for category match, and if none is found it fallbacks to default questions.

## Method **BuildAPIUrl()**:

```
1 string BuildAPIUrl()
2 {
3     string url = apiUrl;
4
5     if (!string.IsNullOrEmpty(selectedDifficulty))
6     {
7         url += $"&difficulty={selectedDifficulty}";
8     }
9
10    if (!string.IsNullOrEmpty(selectedCategory) && selectedCategory != "All Categories")
11    {
12        // Find the category ID from dynamically loaded categories
13        var category = availableCategories.FirstOrDefault(c => c.Value.ToString() == selectedCategory);
14        if (availableCategories.ContainsKey(selectedCategory))
15        {
16            int categoryId = availableCategories[selectedCategory];
17            url += $"&category={categoryId}";
18        }
19        else
20        {
21            // Debug.LogWarning($"Category '{selectedCategory}' not found in available categories. Using all categories.");
22        }
23    }
24
25    // Debug.Log($"API URL: {url}");
26    return url;
27 }
```

Figure 6: Snippet from building the API url

Explanation: The method defines the logic for getting the API url, fetch it with the selected category and difficulty and returns it.

## Method **FetchQuestionsFromAPI()**:

```

1  IEnumerator FetchQuestionsFromAPI()
2  {
3      isLoading = true;
4      ShowLoadingOverLay();
5
6      string url = BuildAPIUrl();
7
8      using (UnityWebRequest webRequest = UnityWebRequest.Get(url))
9      {
10         yield return webRequest.SendWebRequest();
11
12         if (webRequest.result == UnityWebRequest.Result.Success)
13         {
14             try
15             {
16                 var response = JsonUtility.FromJson<OpenTDBResponse>(webRequest.downloadHandler.text);
17                 if (response.response_code == 0 && response.results != null && response.results.Length > 0)
18                 {
19                     ConvertAPIQuestionsToTriviaQuestions(response.results);
20                     // Debug.Log($"Successfully Loaded {response.results.Length} questions from API");
21                 }
22                 else
23                 {
24                     // Debug.LogWarning("API returned no questions, falling back to Local questions");
25                     LoadQuestionsFromJson();
26                 }
27             }
28             catch (Exception ex)
29             {
30                 // Debug.LogError($"Error parsing API response: {ex.Message}");
31                 LoadQuestionsFromJson();
32             }
33         }
34         else
35         {
36             // Debug.LogError($"API Error: {webRequest.error}");
37             LoadQuestionsFromJson();
38         }
39     }
40
41     isLoading = false;
42     HideLoadingOverLay();
43 }
44

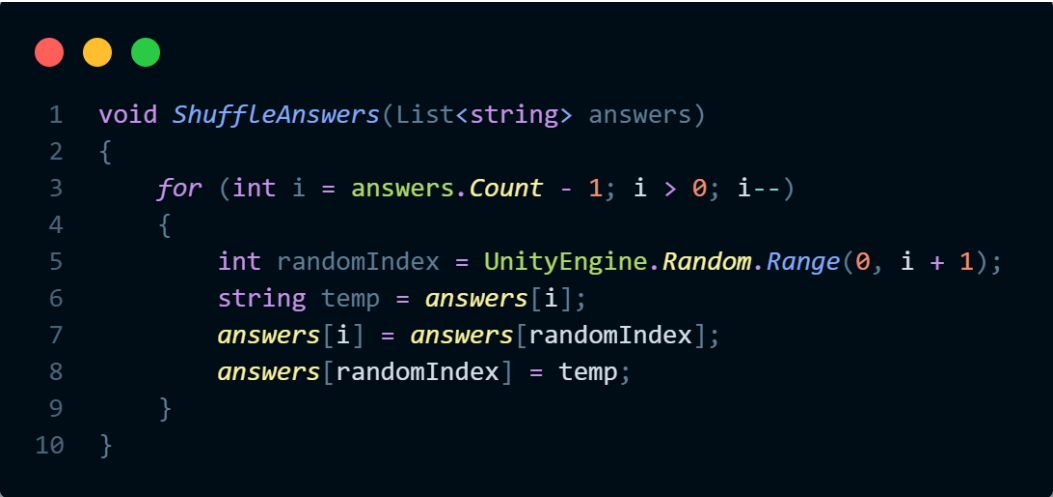
```

Figure 7: Fetching logic for questions

Explanation: Gets the build API, utilizing the Unity module package (UnityWebRequest) for the **get** request. Afterwards wait for a response from the sent request and once the response is received the data is being deserialized into readable format (text) with the implementation of the custome method **ConvertAPIQuestionsToTriviaQuestions()**.

Method **ShuffleAnswers(params)**:





```

1  void ShuffleAnswers(List<string> answers)
2  {
3      for (int i = answers.Count - 1; i > 0; i--)
4      {
5          int randomIndex = UnityEngine.Random.Range(0, i + 1);
6          string temp = answers[i];
7          answers[i] = answers[randomIndex];
8          answers[randomIndex] = temp;
9      }
10 }

```

Figure 8: Shuffle logic for answers

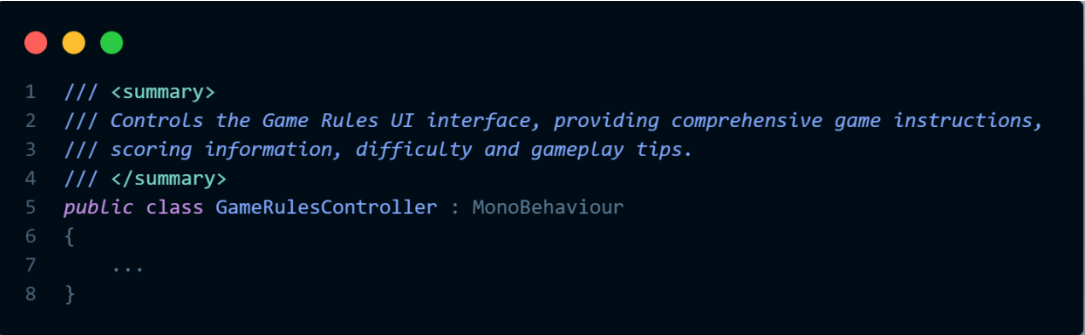
Explanation: The answers are stored in a structured list and an iteration over its elements in reversed order is performed, while a random index is chosen, then a swapping is performed (getting a random index from the iteration of the collection and with the help of third variable an element is append to a random place). This cycle continuous until the last element of the collection.

## VI. Development methodology

### A. Code documentation approach

An inline documentation approach was pursued (the project is documented using comments for each section of the codebase). The commenting style consists of *double slash quotes* (`///`) and *summary tags* (`<summary>` explanation `</summary>`).

Example:



```

1  /// <summary>
2  /// Controls the Game Rules UI interface, providing comprehensive game instructions,
3  /// scoring information, difficulty and gameplay tips.
4  /// </summary>
5  public class GameRulesController : MonoBehaviour
6  {
7      ...
8  }

```

Figure 9: An example of commenting with summary tag

## VII. AI utilization

AI-Assisted Development Tools	Purpose
Unity Assistant	Checking for specific Unity packages, libraries; unit testing
Figma AI	Accelerate prototyping
Copilot	Summarizing articles; web resources
Notion AI	Managing project organization; planning
DeepSeek	Generating fragments of code and explaining parts of code
Manus	Data visualization

### A. AI Limitations and Human Oversight

At some point AI would make errors, so it was necessary to check the provided output whether it will be code (checking for syntax errors, misused libraries), explanations, external validation has had to be done. It consisted of comparing the received information with prior knowledge and external resources like books, articles...

## VIII Conclusion

The development of the Trivia Game project followed a structured approach, leveraging modern technologies and integrating AI-assisted tools throughout its lifecycle. A notable aspect of the process was the **iterative design phase**, particularly evident in the presentation of game rules. Initially planned to appear alongside the game menu, this feature was refined into a dedicated, tabbed window, enhancing user experience and accessibility.

**Opportunities for improvements:**

Creating an AI Validation Process: Although human oversight of AI was occurring, creating a more formalized process for validating AI outputs could be even more efficient; providing an ongoing level of quality control.