Barbecue Party Project Report

In this case study, the goal was to create an endless loop of simple slot game variation that was supported with nice animations and smooth transitions. The main approach of mine was to write cleanest and most extendable code possible. To do that, my first step was creating a MVVM-like template implementation. Since case study is not too complex on the roulette view side, I did not create a RouletteView script. Rather, I decided to handle most of the visual related stuff on slot script because they are the main components that the user interacts with.

To make the system modular, I used scriptable objects and addressables together especially on *reward configs*. Since given assets are just some visuals that need to be shown to the user, I could create a basic enum class to store these rewards as scriptable objects.

Since the company's main work is very similar to the casino games genre in app markets, I wanted to use my prior Unity Cloud and Javascript knowledge to securely implement rarity based outcome decider endpoints and manage crucial aspects of the game from Unity Cloud. For this context, I created 3 different scripts that serve as endpoints for my game.

Decide Outcome:
This script is the main part of the game. It decides which reward is granted to the user when the user pressed the spin button. It does that by checking already claimed rewards of users and possible rewards stored in Unity's remote config and decides reward outcomes from unclaimed rewards for specific users. It decides outcome by calculating total rarity of available rewards and map each reward's probability of occurrence. Then choose a random float and decide the outcome. I am presenting the script below since it is visible only for me in Unity Cloud.

```
const RewardType = {
  Item0: 0,
  Item1: 1,
  Item2: 2,
  Item3: 3,
  Item4: 4,
  Item5: 5,
  Item6: 6,
  Item7: 7,
  Item8: 8,
  Item9: 9,
  Item10: 10,
  Item11: 11,
  Item12: 12,
  Item13: 13
};

const { SettingsApi } = require("@unity-services/remote-config-1.1");
const { DataApi } = require("@unity-services/cloud-save-1.0");

module.exports = async ({ params, context, logger }) => {
  const { projectId, playerId } = context;
  const remoteConfig = new SettingsApi(context);
  const cloudSave = new DataApi(context);
  const services = { remoteConfig, cloudSave, logger };

  if (!playerId) {
    return { error: "Player ID is required but was missing." };
  }

  try {
    const rewardsData = await getRemoteConfigData(projectId, services, "Rewards");

    const receivedRewards = await getPlayerReceivedRewards(projectId, playerId, services) || [];
    const rewardsList = parseRewards(rewardsData).filter(item => !receivedRewards.includes(RewardType[item.name]));
    if (rewardsList.length === 0) {
      await resetClaimedRewards(projectId, playerId, services);
      rewardsList.push(...parseRewards(rewardsData));
    }

    const totalRarity = rewardsList.reduce((sum, item) => sum + item.rarity, 0);
    const cumulativeRanges = [];
    let cumulativeSum = 0;
    rewardsList.forEach(item => {
      cumulativeSum += item.rarity / totalRarity;
      cumulativeRanges.push({ name: item.name, probability: cumulativeSum });
    });
    const random = Math.random();
    let selectedItem = null;
    for (const item of cumulativeRanges) {
      if (random <= item.probability) {
        selectedItem = item.name;
        break;
      }
    }

    const outcomeValue = RewardType[selectedItem];
```

```
    if (outcomeValue === undefined) {
      throw new Error(`Selected item '${selectedItem}' does not match any RewardType.`);
    }
    await savePlayerReceivedReward(projectId, playerId, services, outcomeValue);

    return {
      outcome: outcomeValue
    };

  } catch (error) {
    return { error: `Failed to process rewards data: ${error.message}` };
  }
};

async function getRemoteConfigData(projectId, services, key) {
  const result = await services.remoteConfig.assignSettingsGet(projectId);
  return result.data.configs.settings[key];
}

async function getPlayerReceivedRewards(projectId, playerId, services) {
  try {
    const result = await services.cloudSave.getItems(projectId, playerId, "ReceivedRewards");
    const rewards = result.data.results[0]?.value || [];
    return Array.isArray(rewards) ? rewards : [];
  } catch (error) {
    throw new Error(`Error fetching received rewards for player '${playerId}': ${error.message}`);
  }
}

async function savePlayerReceivedReward(projectId, playerId, services, rewardValue) {
  try {
    const existingRewards = await getPlayerReceivedRewards(projectId, playerId, services) || [];
    existingRewards.push(rewardValue);

    await services.cloudSave.setItem(projectId, playerId, {
      key: "ReceivedRewards",
      value: existingRewards
    });
  } catch (error) {
    throw new Error(`Error saving received reward for player '${playerId}': ${error.message}` );
  }
}

async function resetClaimedRewards(projectId, playerId, services) {
  await services.cloudSave.setItem(projectId, playerId, {
    key: "ReceivedRewards",
    value: [] // Reset to empty array
  });
}

// Parse rewards from remote config data
function parseRewards(rewardsData) {
  if (rewardsData && rewardsData.rewards) {
    return rewardsData.rewards.map(reward => ({
      name: reward.enumName,
      rarity: reward.rarityValue
    }));
  }
  return [];
}
```

Other 2 scripts are: *GetClaimedRewards* and *IsRoundFinished.*

GetClaimedRewards: purpose is to use Unity's cloud save similar to player prefs. It holds current round's claimed rewards so even if a user exits from the app and rejoins, the system keeps track of claimed rewards and shows slots as claimed or available accordingly.

IsRoundFinished: Serves to a similar goal, it checks whether the user claimed all possible rewards from the minigame. If so, it redirects users to the Initial scene of the game.

I also created and implemented a simple Wallet system that can keep track of claimed cumulative rewards for users. I use Player prefs for that purpose. To support the claim feeling, I implemented a simple Wallet trail object. This object is shown to the user as it is the outcome reward of the spin. I configured a very basic particle system to copy the visual and feeling of the trail object of reference videos.

Throughout the case study, I progressively refactored the code for readability and extendability. I tried my best to make tweens modifiable in all aspects from the editor.

The case was an opportunity for me to search and learn Unity's Addressables system. I implemented a simple SceneLoader that loads addressable scenes and also configured an initiating mechanism that waits for rewards and slots to be loaded in the correct order.

This was a great opportunity to show my Unity and coding skills. I appreciate the company for giving me this opportunity to showcase my skills and further develop myself.