Task 1:

-Reduce the batches:

- creating an atlas image using Unity Atlas Packer v2. To achieve this,I used the AtlasImage-1.0.0(https://github.com/mob-sakai/AtlasImage (https://github.com/mob-sakai/AtlasImage)) package, which allows us to select sprites from the atlas in the Unity editor. Additionally, I implement a SpriteManager class to handle the retrieval of sprites for mission and reward icons.

- also I use Enable GPU Instancing feature in Materials .

https://drive.google.com/file/d/1lTeV1oDMe_Suz6LsGOtXq591mpa21mKB/view?usp=drive_link (https://drive.google.com/file/d/1lTeV1oDMe_Suz6LsGOtXq591mpa21mKB/view?usp=drive_link)

- Pooling System:

I Implement a generic pooling system for my Unity project to boost performance and streamline development. By reusing objects instead of constant instantiation and destruction, it minimizes overhead, enhancing runtime efficiency. Its versatility allows pooling for various MonoBehaviour types, offering a clean and maintainable solution. The centralized management and features like cross-scene persistence and dynamic pool size adjustments bring scalability, making game development more enjoyable and efficient.

PoolingSystemManager class:

This class is a singleton that manages object pools. It provides methods to get or create a pool for a given prototype, get a pooled object, and return an object to the pool.

```
// Singleton that manages object pools
public class PoolingSystemManager : MonoBehaviour
{
    // Singleton instance
    static PoolingSystemManager instance;

    // Dictionary to store object pools
    private Dictionary<Type, object> _pools = new Dictionary<Type, object>();

    // Get or create a pool for a given prototype
    public ObjectPool<T> GetOrCreatePool<T>(T prototype, int initialSize = 100) where T : MonoBehaviour
    {
        // ...
    }

    // Get a pooled object
    public T GetPooledObject<T>(T prototype, Vector3 position, Quaternion rotation, Transform parent = null) where T : MonoBehaviour
    {
        // ...
    }
}
```

ObjectPool class :

This class represents an object pool. It stores a list of pooled objects and provides methods to ensure a certain quantity of objects, get a pooled object, and return an object to the pool.

```
// Represents an object pool
public class ObjectPool<T> where T : MonoBehaviour
{
        // List of pooled objects
        private List<T> pool = new List<T>();

        // Ensure a certain quantity of objects
        public void EnsureQuantity(T prototype, int count, Transform parent = null)
        {
            // ...
        }

        // Get a pooled object
        public T GetPooled(T prototype, Vector3 position, Quaternion rotation, Transform parent = null)
        {
            // ...
        }

        // Return an object to the pool
        public void ReturnToPool(T obj)
        {
            // ...
        }

        // Return all objects to the pool
        public void ReturnAllToPool()
        {
            // ...
        }
}
```

Task 2:

-Mission System:

Mission System with Reward System - Design Overview

Introduction

The Mission System with Reward System is a comprehensive architecture designed for Unity, incorporating an observer pattern, MVC (Model-View-Controller) architecture, and Singleton pattern. The system provides a flexible and extensible framework for managing missions, rewards, and user interfaces.

Design Principles

Observer Pattern with EventManager

The system employs an observer pattern facilitated by the `EventManager`. This pattern promotes loose coupling, allowing different components to communicate without direct dependencies. Events, such as mission completion and reward reception, trigger actions across the system.

MVC Architecture

The design adheres to the MVC pattern, distributing responsibilities among the `MissionManager` (controller), various mission classes (models), and `MissionUI` (view). This separation of concerns enhances code organization and readability.

Singleton Pattern

Critical components, including `MissionManager`, `RewardManager`, and `MissionUI`, follow the Singleton pattern. This ensures there is only one instance of each class, facilitating global state management and preventing unnecessary instantiations.

Unity Editor Integration

Unity-specific features, such as `[SerializeField]` attributes, enable seamless integration with the Unity Editor. Mission and reward data can be easily created and modified within the editor, enhancing the development workflow.

Key Components

MissionManager

The central controller manages active missions, handles mission completion, and dynamically loads mission data. It utilizes a factory method for mission creation, making it extensible to new mission types.

RewardManager

Responsible for managing rewards, the `RewardManager` initializes and tracks a list of available rewards. It responds to mission completion events and grants rewards accordingly.

EventManager

A robust event system provides a decoupled communication mechanism. Subscriptions, broadcasts, and cleanup operations are handled, promoting modularity and ease of extension.

MissionUI

The view component displays active missions through UI elements. It subscribes to events, updates mission data, and ensures a responsive user interface.

Code Quality and Best Practices

The code exhibits high quality and adherence to Unity best practices:

- **Readability:** The code is well-structured, with clear comments and consistent naming conventions, ensuring readability.
- **Error Handling:** Robust error handling is implemented, addressing potential issues, such as null or empty mission data.
- **Unity Best Practices:** The implementation follows Unity-specific best practices, leveraging features like `[SerializeField]` and Unity events.

Conclusion

The Mission System with Reward System stands out for its flexibility, maintainability, and scalability. By embracing observer patterns, MVC architecture, and Unity-specific features, it provides a solid foundation for diverse projects. The code quality and design choices contribute to a system that is not only efficient but also easy to extend and maintain over time.

Extension: Dynamically Generating Reward Classes and Enum Values

To enhance the Mission System with Reward System, a dynamic class and enum generation feature has been implemented. This allows for easy adaptation to changes in the `RewardTypeEnum` by auto-generating new reward classes and updating the `RewardManager` through Unity Editor code.

Dynamic Class Generation

A `RewardClassGenerator` has been introduced to automatically create new reward classes based on changes in the `RewardTypeEnum`. This class, when invoked, generates a new reward logic class extending the specified base class. The generated class contains abstract methods that need to be implemented for each reward type.

```
public static class RewardClassGenerator
{
    // ... (previously provided methods)

    public static void GenerateClass(string className, Type baseClassType)
    {
        // ... (previously provided code)

        // Get abstract method names from the base class
        List<MethodInfo> methodNames = GetAbstractMethodNames(baseClassType);

        // Generate abstract methods
        for (int i = 0; i < methodNames.Count; i++)
        {
            code += $"      public override {methodNames[i].ReturnType.Name.ToLower()} {methodNames[i].Name}()" +
                    "\n      {\n         throw new NotImplementedException();\n         }";
        }

        code += "\n   }\n}\n";

        return code;
    }

    // ... (previously provided methods)
}
```

Dynamic Enum Generation

An `EnumBuilder` class has been implemented to dynamically create and manage enums based on changes to the `RewardTypeEnum`. This class includes methods to build an enum, generate its code block, and find the enum block within script contents.

```
public static class EnumBuilder
{
    // ... (previously provided methods)

    public static string GenerateEnumBlock(string enumName, string[] enumValues)
    {
        // ... (previously provided code)
    }

    public static string FindEnumBlock(string scriptContents, string enumName)
    {
        // ... (previously provided code)
    }
}
```

Enum Management

The `RewardEnumManager` scriptable object has been introduced to store the `RewardTypeEnum`. This object allows for easy modification within the Unity Editor, providing a central location to manage the reward enum.

```
[CreateAssetMenu(fileName = "RewardList", menuName = "MissionsData/Reward List", order = 1)]
public class RewardEnumManager : ScriptableObject
{
    public Enum RewardTypeEnum;
}
```

Unity Editor Integration

The generated classes and enums can be managed within the Unity Editor. By changing the `RewardTypeEnum` in the `RewardEnumManager` asset, new classes and enums are auto-generated. Additionally, the `RewardManager` has been updated through editor code to include logic for the newly generated reward classes.

Conclusion

The dynamic generation of reward classes and enums adds a powerful extension to the Mission System with Reward System. This feature allows developers to easily adapt to changes in the `RewardTypeEnum` by automatically generating the required logic. The Unity Editor integration streamlines the process of managing enums and ensures that the `RewardManager` stays up-to-date with the evolving system. This design choice enhances flexibility, maintainability, and scalability, making the system robust and adaptable to future changes.

Generate Mission

*Add New Mission

https://drive.google.com/file/d/11Y9nL7afr0JvUQbDopEz6QTroQHNg0W_/view?usp=drive_link (https://drive.google.com/file/d/11Y9nL7afr0JvUQbDopEz6QTroQHNg0W_/view?usp=drive_link)

*Add New Reward Type with Icon of it.

https://drive.google.com/file/d/1VZj6XwU1HqtMMce5vX-ODv1wBWZgHkI6/view?usp=drive_link (https://drive.google.com/file/d/1VZj6XwU1HqtMMce5vX-ODv1wBWZgHkI6/view?usp=drive_link)

***on mission load only select one of each difficulty and after done all of mission start from beginning.

https://drive.google.com/file/d/13ZrxU0YSNn9WFwAqKGRhW9vxgOw2N6ML/view?usp=drive_link (https://drive.google.com/file/d/13ZrxU0YSNn9WFwAqKGRhW9vxgOw2N6ML/view?usp=drive_link)

*also I fixed the Build Tower in Editor

https://drive.google.com/file/d/1elS9yZlG9jfTgtU1fRwoUUm9l6Nu1gIQ/view?usp=drive_link (https://drive.google.com/file/d/1elS9yZlG9jfTgtU1fRwoUUm9l6Nu1gIQ/view?usp=drive_link)