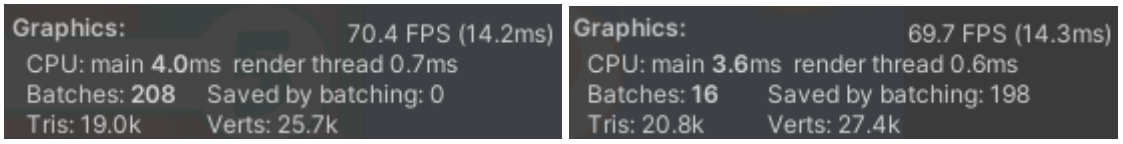
Marc Castells - Homa case study

#1. Implement two optimizations on the project. Even though the project runs smoothly, make two classic optimizations.

Optimize draw calls:



As we can see in the before and after, I reduced the number of batches from 208 to 16. This was achieved by enabling GPU instancing for Exploding, Cylinder and Water materials. Additionally, we created a texture atlas for the UI elements to further reduce the number of batches.

Add a pooling system for the barrels

I created `BarrelPool.cs`, which implements a pooling system similar to how `FxPool.cs` works. The only notable difference is the creation of a method to return a barrel to the pool when it's no longer needed, and a method to return all barrels to the pool. This methods are `ReturnToPool(TowerTile tileToReturn)` and `ReturnAllToPool()` respectively.

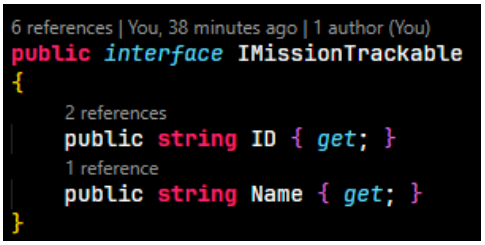
#2. We would like to add missions to the game in order to improve the retention of our players

The mission system consists of `MissionManager.cs`, `MissionManagerData.cs`, `IMission.cs` and `IMissionTrackable.cs`. `MissionManager` is a singleton like `GameManager` and `TileColorManager`. It has the function of refilling the missions whenever necessary and updating their progress. It exposes three events to notify updates, these are:

```
2 references
public event Action<IMission[]> MissionsRefilledEvent;
2 references
public event Action<IMission[]> MissionsUpdatedEvent;
2 references
public event Action<IMission> MissionCompletedEvent;
```

By subscribing to these events we can grant rewards, update the UI, etc. without having to use any game specific code in the manager. This facilitates it's portability to other projects.

In order to know what can be a mission objective without having to rely on any game specific code, I have created `IMissionTrackable` interface. This is what it looks like:



To make any gameplay element of a game an objective, we simply need to make the element implement this interface. Then, when that element is destroyed/collected, we update the missions via the `UpdateMission` method in the mission manager. If any of the currently active missions has an objective that matches, it will be updated accordingly.

`IMission` is an abstract scriptable object that is used to define what a mission is. In order to create a mission, you can simply create a new game specific class that inherits for `IMission` and override its abstract properties and methods. In this project I created `MissionObjectiveTileCollect` from which I created the `TileCollectMissionObjective` and `ExplodingTileCollectMissionObjective` scriptable objects. These scriptable objects can be put in the `missionList` field in the mission manager, and they will then become part of the pool to select potential missions when refilling.

To prevent the game from wiping any mission when reloading the scene, I have created `MissionManagerData`, which holds the currently active missions.

When a mission is completed, the `MissionCompletedEvent` event will be fired. The game can then listen and grant rewards base on the `MIssionReward` scriptable object the mission contains. To create a new type of reward, one simply has to create a new Scriptable object from `MIssionReward`. In this specific game, there aren't any resources one might give as a reward, so I created a hypothetical reward called `CoinMissionReward`. Because the game doesn't any resources, a reward will simply be logged in the console to show the functionality.

As we can see, this mission system is designed to be as minimally coupled with any project as possible. With this design, the game only has to make the desired objective implement `IMissionTrackable` and create the desired missions and rewards. Then it's just a matter of hooking the mission update and mission complete functionality.

I also added a `MISSIONS_ENABLED` bool in `RemoteConfig.cs`, and implemented the necessary logic to deactivate the mission system (both UI and logic) if the bool is set to false.

The mission UI can be accessed through the missions button in the top left corner of the screen. When clicked, the button will show the currently active missions and their progress. The missions UI can be accessed while playing, but if we wanted to hide it during gameplay, we could activate/deactivate the button through the animator, like the options button.