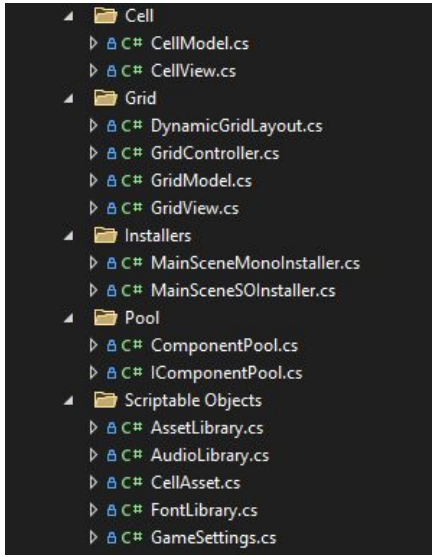
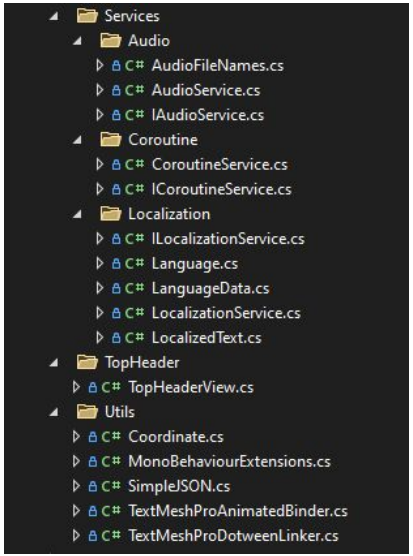
BASIC
MATCH 3 GAME

PROJECT OVERVIEW

- Respecting the single responsibility principle and a basic model-view-controller architecture

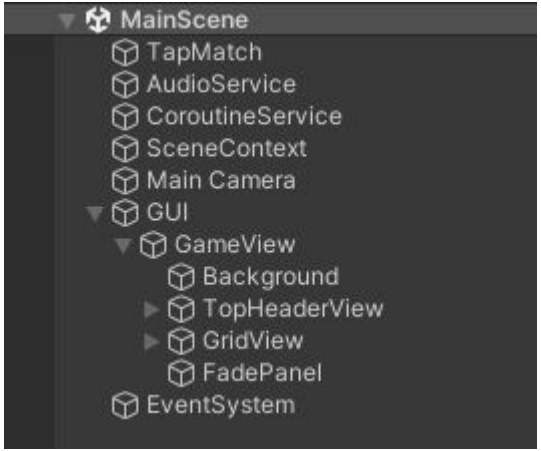
(also my scripts will never have more than 200 lines of code, team clean code yes)





SCENE HIERARCHY

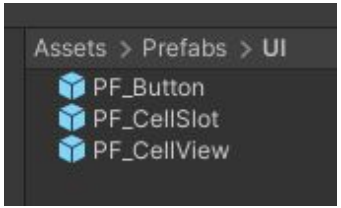
- Clean structure for easy modification



PREFABS MANAGEMENT

- Prefabs are key to avoid scene conflicts and for designers to test/make updates without interfering in the programmers work!

(Although merge conflicts will eventually appear hehe)



GRID BUILDING PROCESS

1. Fill the grid with random matchables
2. When you tap on a matchable recursively iterate its adjacent neighbors and so on checking the type is the same

GRID BUILDING PROCESS

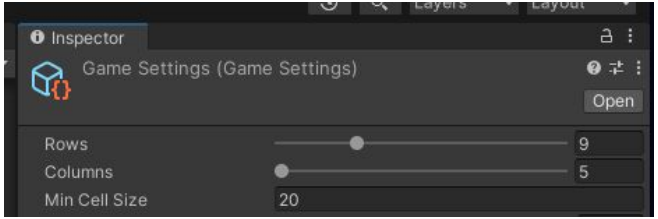
3. Empty connected cells
4. Fill bottom empty cells, by shifting cells downwards from top

GRID BUILDING PROCESS

5. Fill the remaining slots at the top with new random matchables (I only fill slots of columns where there's at least 1 empty slot)
6. Rinse and repeat, no turn limits (canvas group interactability is disabled until grid is filled again, so player cannot tap until it's done)

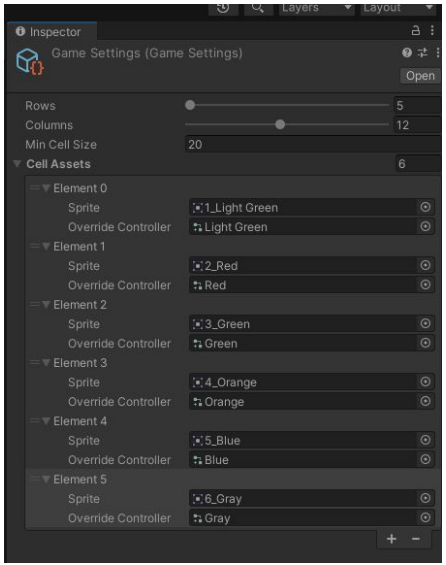
GRID BUILDING PROCESS

Grid size easily customizable via Game Settings Scriptable Object (cannot go outside specified range of 5-20, Editor Script)



GRID BUILDING PROCESS

For the matchables appearance, in the same Scriptable Object (cannot go outside specified range of 3-6, Editor Script)



MY ROUTINE WORK PROCESS

UNDERSTAND REQUIREMENTS

MAKE IT WORK

CLEAN AND OPTIMIZE IT

PROJECT FEATURES

- Match 3
- Animations
- Music & Sound FX
- Language Change Button
- Bomb Button
- Dependency Injection
- Unit Tests

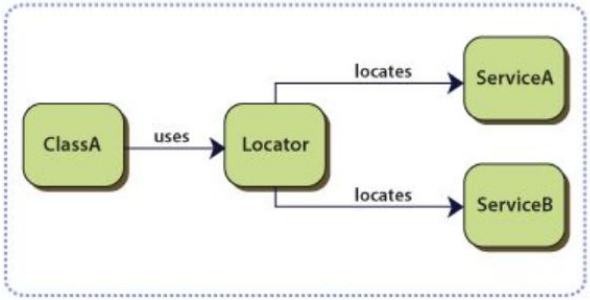
DEPENDENCY INJECTION

Why I’m not a big fan of **singletons**?

- Because you are limited to one instance
  (e.g. Imagine that in January the requirements are to have 1 avatar replayer, but then in May they change to 3 avatar replayers per scene, you are screwed my friend…)
- Because they are public to everyone
  (This can lead to misuse, specially for junior devs.)
- Because they are immutable
  (e.g. I have a singleton that does X, but on some cases I want to do Y, so I have to constantly change it.)
- Refactoring a singleton is painful
  (If you need to manually change every “.Instance” in your code base for another class “.Instance”, it could take years to complete the refactor…)

DEPENDENCY INJECTION

You can still use the Service Locator pattern (which I commonly use), but for me it's still the same deal, a singleton of singletons



DEPENDENCY INJECTION

What do I prefer to do instead?

- Program to an interface and use a dependency injection framework
  (either code your own or use an existing one)

DEPENDENCY INJECTION

❌ NOT RECOMMENDED WAY

```
Unity Script (1 asset reference) | 0 references
public class Player : MonoBehaviour
{
    0 references
    public void OnPlayerHit()
    {
        // immutable implementations
        EventSystemManager.Instance.TriggerEvent("PlayerHit");
        AudioPlayingManager.Instance.PlayAudioClip("PlayerHitFx");
        // ...
    }
}

Unity Script (1 asset reference) | 0 references
public class Button : MonoBehaviour
{
    0 references
    public void OnPress()
    {
        // immutable implementation
        AudioPlayingManager.Instance.PlayAudioClip("ButtonPressFx");
    }
}
```

DEPENDENCY INJECTION

✔ RECOMMENDED WAY

```
Unity Script (1 asset reference) | 0 references
public class Player : MonoBehaviour
{
    [Inject] private IEventsSystemService _eventsSystemService;
    [Inject] private IAudioPlayingService _audioPlayingService;

    0 references
    public void OnPlayerHit()
    {
        _eventsSystemService.TriggerEvent("PlayerHit");
        _audioPlayingService.PlayAudioClip("PlayerHitFx");
        // ...
    }
}
```

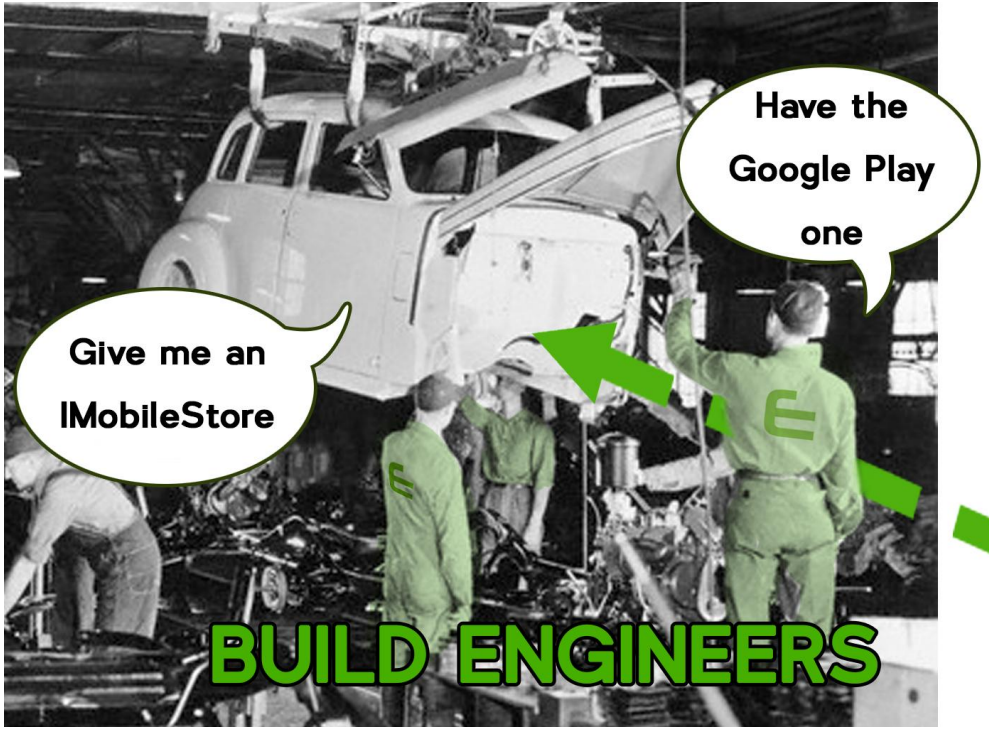
```
Unity Script (1 asset reference) | 0 references
public class Button : MonoBehaviour
{
    [Inject]
    private IAudioPlayingService _audioPlayingService;

    0 references
    public void OnPress()
    {
        _audioPlayingService.PlayAudioClip("ButtonPressFx");
    }
}
```

```
Unity Script (1 asset reference) | 0 references
public class DependencyInjectionInstallerExample : MonoInstaller
{
    9 references
    public override void InstallBindings()
    {
        // With just one line of code you can change all the dependencies from the code base!
        Container.Bind<IEventsSystemService>().To<DefaultEventsSystemService>().AsSingle();
        Container.Bind<IAudioPlayingService>().To<AudioStorePlayingService>().AsSingle();

        // You can do cool stuff such as:
        #if UNITY_EDITOR
            Container.Bind<IMobileStore>().To<TestMobileStore>().AsSingle();
        #elif UNITY_ANDROID
            Container.Bind<IMobileStore>().To<GooglePlayStore>().AsSingle();
        #elif UNITY_IOS
            Container.Bind<IMobileStore>().To<AppleAppStore>().AsSingle();
        #endif
    }
}
```

DEPENDENCY INJECTION



IMobileStore Implementations:

AmazonAppStore

AppleAppStore

GooglePlayStore

UNIT TESTS

- TDD approach is always a good habit