Notes de cours
Scènes, De/Serialization, et PlayerPrefs

Finalité et apprentissage clés

- Navigation de scène
- Sérialization et Desérialization avec Json.NET
- Préférences avec playerPrefs

Concepts et Navigation de Scènes

Création d'une scène mère

Dans la plupart des cas, nous voulons définir une scène principale d'où notre jeu sera lancé.
Cette scène mère s'occupera de gérer toutes les fonctionnalités qui sont indépendante de la scène sur laquelle on se trouve, comme les références statiques pour la navigation de scène, etc.

De base, tous les objets d'une scène précédente sont détruits lorsqu'on utilise LoadScene(), mais il est possible d'empêcher cela.

1) Dans la fenêtre de projet, allez dans votre dossier **Assets** > **Scenes**, et puis clic droit pour sélectionner l'option **Create** > **Scene**. Et nommez-la « _base », « _start » ou « _source », à votre guise.
2) Assigner les scripts nécessaires à votre jeu sur un objet vide
3) Créez un script **DontDestroyOnLoad.cs** et attachez-le sur ce même objet avec vos scripts.
4) Dans **Awake**(), ajoutez la ligne suivante :

```
DontDestroyOnLoad(this.gameObject);
```

Ceci devrait prévenir la destruction de l'objet et préserver vos scripts qui y sont attachés entre les changements de scène.
Il ne reste plus qu'à s'assurer que votre nouvelle scène soit la première pour le build de votre jeu.

1) Dans **File** > **BuildSettings** > **Scenes In Build**, assurez vous d'ajouter votre scène mère et qu'elle soit la première dans la liste.

Il sera recommandé à partir de maintenant de toujours ouvrir cette scène quand vous voulez tester votre jeu.

Navigation de Scène

Bon! C'est bien beau, mais à partir de notre scène mère nous devrions directement ouvrir notre scène d'acceuil, souvent le menu. C'est pourquoi la scène mère est souvent l'endroit parfait pour situer un script de navigation de scène!

Comme le SceneManager utilise des fonctions statiques, nous pouvons faire de même et simplement déclarer nos fonctions de navigation comme statique sans avoir besoin de déclarer une référence statique d'instance pour notre SceneNavigator.

1) Dans votre scène mère, sous votre objet persistant, créez et assignez un nouveau script appelé **SceneNavigator.cs**
2) Ajoutez la ligne **using UnityEngine.SceneManagement;** dans votre en-tête.
3) Ajoutez vos fonctions statique de navigation comme StartGame(), GoToMenu(), etc.
4) Dans Start(), ouvrez la scène d'acceuil (Menu dans ce cas)

```
using UnityEngine;
using UnityEngine.SceneManagement;

Script Unity | 0 références
public class SceneNavigator : MonoBehaviour
{
    Message Unity | 0 références
    void Start()
    {
        OpenMenu();
    }

    1 référence
    public static void OpenMenu()
    {
        SceneManager.LoadScene("Menu");
    }

    0 références
    public static void OpenGame()
    {
        SceneManager.LoadScene("Game");
    }
}
```

PlayerPrefs

Dans un jeu, nous voulons souvent retenir certaine information sur nos joueurs, comme les préférences linguistiques, les paramètres personnels, etc.
Une interface bien utile pour cela est les PlayerPrefs fournie par Unity!
Il s'agit d'une chose similaire aux cookies dans le développement Web.
Ils seront persistants d'une session de jeu à l'autre et permettra à chaque utilisateur de personnaliser son expérience de façon consistante.

Il est recommandé de créer ses propres helpers et interfaces autour de ceci pour permettre plus de fonctionnalité, mais pour les besoins de la cause nous prendrons une approche toute simple.

1) Dans votre scène mère, sous votre objet persistant, créez et assignez un nouveau script appelé **GameSettings.cs**
2) Pour chaque paramètre que nous voudrons ajouter un wrapper. Alors déclarer un nouveau membre statique comme suis :

```
public class GameSettings : MonoBehaviour
{
    0 références
    public static float SoundVolume
    {
        get => PlayerPrefs.GetFloat("SoundVolume", defaultValue: 1f);
        set => PlayerPrefs.SetFloat("SoundVolume", value);
    }

}
```

3) Assurez-vous d'avoir une clé unique pour chaque paramètres.

Vous pourrez maintenant interagir avec ces valeurs dans vos autres scripts à travers GameSettings.SoundVolume

Comme par exemple, la lire lors de l'initialisation de votre musique en jeu, ou de la modifier avec un slider dans une fenêtre de paramètres

```
Script Unity (1 référence de ressource) | 0 références
public class UserSettingsPanel : MonoBehaviour
{
    public Slider volumeSlider;

    // Start is called before the first frame update
    Message Unity | 0 références
    void Start(){
        volumeSlider.value = UserSettings.SoundVolume;
    }

    0 références
    public void UpdateVolume(float val)
    {
        UserSettings.SoundVolume = val;
    }

    0 références
    public void OpenPanel()
```

Note :
Il existe 3 types de valeurs qui peuvent être enregistrées avec PlayerPrefs :

- float : PlayerPrefs.GetInt()
- int : PlayerPrefs.GetFloat()
- string : PlayerPrefs.GetString()

Si vous avez besoin d'enregistrer une valeur **bool**, je vous recommande d'utiliser un playerPrefs int ainsi :

```
public static bool MuteSound
{
    get => PlayerPrefs.GetInt("MuteSound", defaultValue: 0) == 1 ? true : false;
    set => PlayerPrefs.SetInt("MuteSound", value ? 1 : 0);
}
```

Si vous avez besoin d'enregistrer une valeur **enum**, je vous recommande d'utiliser un playerPrefs int ainsi :

```
0 références
public static GraphicsQuality GraphicsQuality
{
    get => (GraphicsQuality)PlayerPrefs.GetInt("GraphicsQuality", defaultValue: 1);
    set => PlayerPrefs.SetInt("GraphicsQuality", (int)value);
}

2 références
public enum GraphicsQuality
{
    Low = 0,
    Medium = 1,
    High = 2
}
```

Important : Les PlayerPrefs ne sont pas recommandées pour sauvegarder la progression des joueurs!

Sauvegarde et Chargement

Très souvent, vous voudrez sauvegarder la progression de votre joueur, et même permettre d'avoir plusieurs parties en parallèle.
Pour cela, nous allons utiliser la Sérialization et la Désérialisation vers un fichier json grâce à la bibliothèque NewtonSoft!
Encore une fois nous allons, pour le besoin pédagogique, prendre une approche simplifiée, mais cela devrait vous permettre de comprendre les bases pour un jour l'utiliser de façon plus complexe.
Chaque jeu requiert une approche unique quant à la structure des données à sauvegarder.

Nous avons un emplacement unique sur le disque de l'utilisateur : **Application.persistentDataPath**

1) Dans votre scène mère, sous votre objet persistant, créez et assignez un nouveau script appelé **SaveSystem.cs**
2) Dans l'entête, ajoutez using NewtonSoft.json; et using System.IO;
3) En dehors de votre classe, créez une public class GameSave{} avec toutes vos propriétés
   Example :

```
public class GameSave
{
    public int stageProgress; // Le niveau auquel le joueur était rendu
    public float score; // Le score total du joueur
    public string playerName; //Le nom du joueur
}
```

4) Dans SaveSystem, créez une fonction public static SaveGame(GameSave save){}
   Dans cette fonction, nous voudrons sérialiser notre data, puis l'enregistrer dans un fichier texte :

```
public static void SaveGame(GameSave save)
{
    var serializedSave = JsonConvert.SerializeObject(save);

    var path = Path.Combine(Application.persistentDataPath, $"game.save");
    File.WriteAllText(path, serializedSave);
}
```

5) Ajoutez une seconde fonction, public static GameSave CheckAndLoadSaveData(){}, où vous viendrez regarder si une partie Sauvegardée existe et la charger :

```
public static GameSave CheckAndLoadSaveData()
{
    var path = Path.Combine(Application.persistentDataPath, $"game.save");

    if (File.Exists(path)){
        var serializedSave = File.ReadAllText(path);
        return JsonConvert.DeserializeObject<GameSave>(serializedSave);
    }
    else return null;
}
```