Sauvegarder les données de jeu (PlayerPrefs, fichiers locaux)

Introduction

La gestion de la sauvegarde des données de jeu est un aspect crucial pour offrir aux joueurs une expérience fluide et continue. Les joueurs doivent pouvoir revenir à leur progression, quel que soit l'endroit où ils interrompent leur session. Cela nécessite des mécanismes efficaces pour sauvegarder et charger les données liées à la progression du joueur, les paramètres du jeu, et d'autres informations essentielles.

---

1. Sauvegarder les Données de Jeu dans Unity

1.1 PlayerPrefs : Sauvegarde Simples et Rapide

PlayerPrefs est une méthode simple de stockage de données dans Unity, souvent utilisée pour enregistrer des informations liées à la configuration ou aux scores. Il est particulièrement adapté pour des données de type clé-valeur simples.

Caractéristiques de PlayerPrefs :

- **Données sauvegardées sous forme clé-valeur** : Les informations sont stockées comme des paires clé-valeur (ex : "ScoreMax" = 500).
- **Stockage local** : Les données sont stockées dans des fichiers spécifiques à la plateforme. Sur Windows, elles sont stockées dans le registre, tandis que sur macOS, elles sont stockées dans un fichier plist.
- **Types de données supportés** : PlayerPrefs peut sauvegarder des chaînes de caractères, des entiers et des flottants.

Utilisation de PlayerPrefs :

- **Enregistrer une donnée :**

```
PlayerPrefs.SetInt("HighScore", 1000);  // Sauvegarder u
n score
PlayerPrefs.SetString("PlayerName", "John");  // Sauvega
rder un nom
PlayerPrefs.Save();  // Sauvegarde explicite des données
```

- **Charger une donnée :**

```
int highScore = PlayerPrefs.GetInt("HighScore", 0);  //
Récupère le score, 0 étant la valeur par défaut
string playerName = PlayerPrefs.GetString("PlayerName",
"Guest");  // Récupère le nom du joueur
```

- **Effacer des données :**

```
PlayerPrefs.DeleteKey("PlayerName");  // Supprimer une c
lé spécifique
PlayerPrefs.DeleteAll();  // Supprimer toutes les donnée
s sauvegardées
```

Limites de PlayerPrefs :

- **Capacité de stockage limitée** : PlayerPrefs est conçu pour des données légères et peut ne pas être adapté pour des informations volumineuses.
- **Pas de support pour des données complexes** (comme des tableaux ou des objets personnalisés).

---

2. Sauvegarder avec des Fichiers Locaux

Lorsque les données à sauvegarder sont plus complexes ou volumineuses, il est préférable d'utiliser des fichiers locaux. Ces fichiers peuvent être au format texte (JSON, XML, etc.), binaire ou même personnalisé. Cela permet de stocker des structures de données plus complexes telles que des inventaires, des niveaux débloqués, ou même des sauvegardes complètes de jeu.

2.1 Sauvegarder des données en JSON

Le format JSON (JavaScript Object Notation) est un moyen courant de sauvegarder et de charger des données dans Unity. Il est léger, facile à utiliser et peut stocker des données complexes sous forme de chaînes.

Exemple : Sauvegarde d'un objet avec JSON

- **Créer une classe de données :**

```
[System.Serializable]
public class PlayerData
{
    public int score;
    public string playerName;
}
```

- **Sauvegarder les données dans un fichier JSON :**

```
using System.IO;
using UnityEngine;

public class SaveManager : MonoBehaviour
{
    public PlayerData playerData;

    public void SaveData()
    {
        string path = Application.persistentDataPath +
"/playerData.json";  // Chemin du fichier
        string json = JsonUtility.ToJson(playerData);
// Convertir les données en JSON
        File.WriteAllText(path, json);  // Sauvegarder d
ans un fichier
    }
}
```

- **Charger les données depuis le fichier JSON :**

```
public void LoadData()
{
```

```
    string path = Application.persistentDataPath + "/pla
yerData.json";  // Chemin du fichier
    if (File.Exists(path))
    {
        string json = File.ReadAllText(path);  // Lire l
e contenu du fichier
        playerData = JsonUtility.FromJson<PlayerData>(js
on);  // Convertir le JSON en objet
    }
    else
    {
        Debug.LogWarning("Fichier de sauvegarde introuva
ble");
    }
}
```

2.2 Sauvegarder des données en Binaire

Le format binaire est plus efficace pour sauvegarder des données complexes ou volumineuses, mais il est moins lisible par l'homme.

Exemple : Sauvegarde d’un objet avec un fichier binaire :

- **Sauvegarde avec un fichier binaire :**

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;

public class BinarySaveManager : MonoBehaviour
{
    public PlayerData playerData;

    public void SaveData()
    {
        string path = Application.persistentDataPath +
"/playerData.dat";  // Chemin du fichier
        BinaryFormatter formatter = new BinaryFormatter
();
```

```
        FileStream file = File.Create(path);
        formatter.Serialize(file, playerData);  // Séria
lisation de l'objet
        file.Close();
    }
}
```

- **Charger avec un fichier binaire :**

```
public void LoadData()
{
    string path = Application.persistentDataPath + "/pla
yerData.dat";  // Chemin du fichier
    if (File.Exists(path))
    {
        BinaryFormatter formatter = new BinaryFormatter
();
        FileStream file = File.Open(path, FileMode.Ope
n);
        playerData = (PlayerData)formatter.Deserialize(f
ile);  // Désérialisation de l'objet
        file.Close();
    }
    else
    {
        Debug.LogWarning("Fichier de sauvegarde introuva
ble");
    }
}
```

2.3 Comparaison entre JSON et Binaire

- **JSON** : Plus lisible et compatible avec d'autres plateformes et langages. Idéal pour les petites et moyennes quantités de données.
- **Binaire** : Plus compact et performant pour les grandes quantités de données. Moins lisible, mais plus rapide à charger.

---

3. Gestion des Fichiers Locaux : Sécurité et Performance

3.1 Sécuriser les Données

- **Cryptage** : Si les données sauvegardées sont sensibles (comme des informations personnelles), il est important de les crypter avant de les écrire dans des fichiers.
- **Validation des données** : Lors du chargement des fichiers, il est essentiel de vérifier leur intégrité pour éviter les corruptions ou erreurs de lecture.

3.2 Performance

- **Réduire les appels de sauvegarde fréquents** : Sauvegarder trop souvent peut nuire à la performance du jeu, notamment dans des jeux à grande échelle.
- **Compression des données** : Pour les grandes quantités de données, la compression (par exemple, en utilisant des fichiers `.zip`) peut réduire l'impact sur les performances.

Conclusion

La gestion des sauvegardes dans les jeux vidéo est essentielle pour permettre aux joueurs de conserver leur progression. En Unity, vous pouvez utiliser **PlayerPrefs** pour des données simples, ou des **fichiers locaux** (JSON, binaire) pour des données plus complexes. Chaque méthode a ses avantages et limites, et le choix dépend de la taille, de la complexité des données à sauvegarder et de la plateforme cible. Les bonnes pratiques de gestion des données, comme le cryptage et la validation, assurent la sécurité et la performance des sauvegardes dans vos jeux.