Roll a ball
mécaniques avancées

1 - GameManager - Pattern Singleton

Lors du développement d'une application, il est parfois utile d'avoir une instance de classe qui sera accessible depuis l'ensemble du code. On utilise alors un singleton.
Attention à ne pas en abuser, trop de singletons peuvent rendre le code difficile à maintenir.

- Créer un nouveau script, le nommer GameManager.
- Créer un GameObject "GameManager" vide dans la hiérarchie et y glisser le script GameManager.
- Dans ce script, déclarer une variable "public static GameManager Manager;"

Il sera possible d'accéder à cette variable depuis n'importe quel autre script en faisant "GameManager.Manager"

```
public void Awake()
{
    if (GameManager.Manager != null) { Destroy(this); }
    Manager = this;
}
```

Mais avant, il faut s'assurer que cette variable à toujours une valeur. Ici on va l'initialiser dans la fonction Awake de son script.
Il est souvent conseillé de vérifier qu'une instance static n'a pas déjà une valeur avant d'en attribuer une. Dans ce cas, on détruit la nouvelle instance:

Dans le but de garder un code propre, on va déplacer les fonctions dans le scripts les plus appropriés:
Les fonctions qui définissent les mouvement du joueur et ses interactions reste dans le PlayerController.cs
Les fonctions qui définissent l'état général du jeu sont à mettre dans le GameManager.cs.

- Déplacer les fonctions de PlayerController.cs qui ne sont pas directement liées au contrôle du joueur et la détection des collisions dans GameManager.cs. Ici, les fonctions de mise à jour de l'interface, et de fin de partie.
- Ajouter une fonction "public void Collect()" qui sera appelé lorsque le joueur ramasse un collectible. Vous pouvez aussi déplacer la désactivation de celui-ci dans cette fonction. Pour ceux-ci, ajouter un paramètre GameObject afin d'avoir la référence de l'objet à désactiver.

2 - Ajout d'un Timer

- Ajouter un nouvel élément de texte à l'interface. Le déplacer en haut à droite de l'écran.
- Ajouter une référence à cet élément dans le GameManager.

- Ajouter une variable float currentTime pour stocker la durée actuelle de la partie.

Pour la gestion du temps, on utilise la classe Time d'Unity. La propriété Time.deltaTime permet d'obtenir le temps qu'a pris la frame précédente à se calculer.

/!\ Lorsqu'on effectue des calculs dans une fonction update (ex: le déplacement d'un objet) il est important de prendre en compte la valeur de Time.deltaTime afin que celui-ci soit indépendant du temps de calcul des frames dans l'application.

- Dans la fonction update, incrémenter currentTime de Time.deltaTime à chaque frame.
- Mettre à jour l'affichage du temps dans l'interface. Vous pouvez utiliser currentTime.ToString("00.000") pour formater le temps.

3 - Créer une nouvelle zone de jeu

- Créer un nouveau GameObject vide dans la scène et le nommer [Zone1], y mettre tous les éléments de la première zone de jeu. (Sol, mur et collectibles)
- Créer une nouvelle zone de jeu, et la placer à côté de la zone 1. Vous pouvez dupliquer un élément sélectionné dans la hiérarchie avec ctrl+d.

Vous ajouterez plusieurs nouvelles zones par la suite. On va donc chercher une méthode itérative pour parcourir les zones.

- Créer un script ZoneManager.cs l'ajouter au GameObject [Zone1] et [Zone2]
- Dans ce script, déclarer une variable "public int CollectibleCount" pour indiquer le nombre de collectible restant dans cette zone.

On pourra ainsi modifier pour chaque zone le nombre de collectible s'y trouvant directement depuis l'éditeur.

- Déclarer une variable "public GameObject Door" pour indiquer le GameObject à désactiver une fois la zone fini
- Attribuer des valeurs à ces variables depuis l'éditeur d'Unity.
- Ajouter une variable "public List<ZoneManager> Zones" dans le GameManager.

/!\ En c# les listes doivent normalement être initialisé avec maList = new List<T>();
Mais lorsqu'elles sont déclarées en variable publique dans un monoBehaviour, l'éditeur d'unity va se charger de l'instanciation de cette liste.

- Glisser dans la liste Zones du GameManager le ZoneManager de la Zone1 et de la Zone 2.
- Ajouter une variable "public int currentZone" au GameManager afin de savoir dans quelle zone on se trouve actuellement.
- Modifier la fonction collect() afin de décrémenter le CollectibleCount de la zone actuelle lorsqu'un nouveau collectible est ramassé.

Lorsqu'on entre en play mode les monoBehaviour sont instanciés, toutes les modifications qui y sont faites seront donc annulées à la fin de l'exécution, les variables CollectibleCount seront donc réinitialisées à chaque fois.

Pour accéder au ZoneManager de la zone actuelle vous pouvez faire Zones[currentZone]

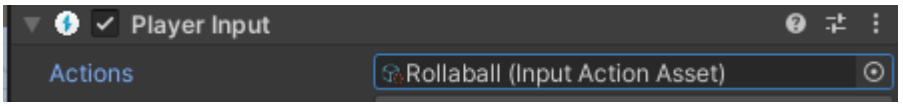
- Ajouter une condition pour tester lorsque le nombre de collectibles de la zone atteint 0. Pour connaître le nombre de collectibles dans une zone, vous pouvez le saisir dans l'éditeur pour chaque Zone manager, ou le calculer par script. (Par exemple en comptant le nombre d'enfants d'un GameObject)
- Désactiver la porte de cette zone à ce moment-là. Et incrémenter la zone actuelle.

4 - Ajouter une zone avec un terrain tournant

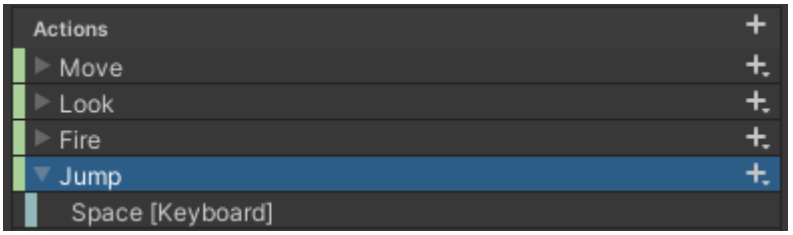
- Créer une nouvelle zone. Dans celle-ci on va supprimer les murs latéraux et allonger le sol afin de créer une passerelle.
- Sur le sol, ajouter un script RotateGround.cs afin de faire tourner le sol dans sa fonction FixedUpdate()

5 - Ajouter une fonctionnalité de saut

- Créer une nouvelle zone. Dans celle-ci on va ajouter un mur à sauter pour le joueur.
- Dans le GameObject Player, dans le composant PlayerInput double cliquer sur l'input action asset afin d'ouvrir sa fenêtre d'édition



- Ajouter une action Jump, et lui définir la touche d'espace.



- Dans le PlayerController.cs, ajouter une fonction "public OnJump(InputValue jumpValue)" qui va ajouter une force de type impulse au joueur sur l'axe vertical.

6 - Ajouter des projectiles

- Créer une nouvelle zone. Dans celle-ci on va ajouter des tours sur les murs qui tirent des projectiles.
- Ajouter un cube, le nommer "Tower". L'orienter avec l'axe z dans la direction du tir. Enlever le BoxCollider du GameObject.
- Créer un prefab projectile: Créer une sphère dans la scène, lui donner la position (0,0,0) et scale (0.2, 0.2, 0.2). Ajouter un Rigidbody, mettre UseGravity à false. Glisser le projectile dans un dossier Prefab. Le supprimer de la scène.
- Créer un script TowerController.cs, et l'ajouter au cube.
- Créer une variable "public float ReloadTime" dans ce script.

Pour la gestion du temps de recharge, on va utiliser des Coroutines.
Les Coroutines sont une fonctionnalité d'unity pour créer des fonctions qui peuvent s'exécuter en plusieurs fois.
Les coroutines peuvent être très pratiques pour exécuter du code en différé.
/!\ Les coroutines sont exécutées comme des Update lors de la boucle d'exécution. Ce n'est pas du multi-thread!

- Déclarer une fonction:

```
public IEnumerator ShootCoroutine(float time)
{
    while (true)
    {
        yield return new WaitForSeconds(time);
        CreateBullet();
    }
}
```

Les coroutines doivent contenir un "yield return". Ce return définit la condition avant de continuer son exécution à partir de cette ligne. Ici, le WaitForSeconds signifie qu'il va attendre que le temps "time" se soit passé avant de reprendre son exécution.
Il est possible de faire un "yield return null" afin de reprendre l'exécution à la frame suivante.

- Pour lancer l'exécution de la coroutine, ajouter "StartCoroutine(ShootCoroutine(ReloadTime))" dans le Start.
- Ajouter une variable public GameObject Bullet, pour faire référence au prefab bullet (à glisser dans l'éditeur). Ainsi qu'une variable ShootSpeed qui définit la vitesse du projectile.
- Créer une fonction

```
private void CreateBullet()
{
    GameObject newBullet = Instantiate(Bullet, transform.position + (transform.forward * 0.6f), Quaternion.identity);
    newBullet.GetComponent<Rigidbody>().AddForce(transform.forward * ShootSpeed,ForceMode.Impulse);
}
```

On va maintenant ajouter la détection de collision sur les projectils. Dès qu'un projectile touche un objet, celui-ci s'auto-détruit. Lorsque le joueur se fait toucher, on va recharger la scène afin de recommencer la partie.

- Dans le GameManager.cs, ajouter une fonction

```
public void RestartGame()
{
    SceneManager.LoadScene(0);
}
```

La fonction LoadScene du scene manager prend soit un entier, soit un string en paramètre. Pour pouvoir être chargée, la scène doit figurer dans la liste des scènes dans build settings: "File/Build settings…"

- Sur le PlayerController.cs, ajouter une fonction

```
public void GotHit()
{
    GameManager.Manager.RestartGame();
}
```

- Créer un script BulletController.cs, avec une fonction

public void OnCollisionEnter(Collision collision)

```
public void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        collision.gameObject.GetComponent<PlayerController>().GotHit();
    }
    Destroy(this.gameObject, 0.1f);
}
```

et l'ajouter sur le prefab Bullet.

8 - Faire tourner les tourelles

L'un des avantages du système de composants, et qu'il permet de facilement cumuler des comportements sur un GameObject.

- Dupliquer le prefab de Tourelle. Sur le nouveau prefab, ajouter le script qui permettait de faire tourner le sol dans la partie 4.

Les projectiles sont enfants des tourelles, ils vont donc en suivre la rotation. Enlever le paramètre transform lors de l'instanciation des projectiles afin qu'ils soient indépendants de la rotation des tourelles.

- Vous pouvez ajouter cette nouvelle version de tourelle à la zone ou créer une nouvelle zone avec.

9 - Plaque de saut

Pour ajouter des plaques de saut, on peut utiliser un cube dont on réduit l'échelle sur l'axe Y. Passer son collider en trigger pour que le joueur puisse aller dessus.
Ajoutez un script sur la plaque qui lorsque le joueur rentre dans le trigger on récupère le rigidbody du Player et on y ajoute une force verticale, similaire à la fonction de saut.

10 - Ajouter une zone avec des ennemis

On va créer une variante de Bullets qui se dirigent vers le joueur lorsqu'il entre dans la zone.
Cette version sera déplacée par script, et non pas par physique. On peut donc enlever le composant rigidbody. On veut aussi que celles-ci touchent le joueur lorsqu'elles entrent en contact avec lui, on peut donc garder le script BulletController.
On va créer et ajouter un script BulletFollow.cs qui va déplacer la sphère vers le joueur lorsqu'il entre dans la zone.
Pour savoir où déplacer les sphères vous pouvez utiliser la fonction Vector3.MoveTowards.

Pour savoir lorsqu'une zone s'active, on va utiliser les UnityEvents. Les UnityEvents sont des implémentations de delegates gérés par Unity. Ils permettent de stocker un ensemble de fonctions qui écoutent un event, elles seront ensuite appelées lorsque l'event est invoqué.
Dans le GameManager,

- ajouter "using UnityEngine.Events;"
- ajouter une variable "public UnityEvent<int> OnNextZone;"
- Dans la fonction NextZone, lorsqu'une nouvelle zone est activée, ajouter "OnNextZone.invoke(currentZoneIndex);"

OnNextZone est un UnityEvent que peuvent écouter des fonctions prenant en paramètre un int, il est possible de passer aucun ou plusieurs types à un Event.
UnityEvent OnEvent; pour aucun paramètre.
UnityEvent<T,X,...> OnEvent pour plus.
Lorsqu'on appel invoke sur un event, toutes les fonctions en listener sont appelé avec le paramètre (ici currentZoneIndex).

Dans le script BulletFollow.cs,

- Ajouter une fonction OnChangeZone(int newZone) {}
- Ajouter dans le Start GameManager.Manager.OnNextZone.AddListener(OnChangeZone);
- Ajouter une variable public int BulletZone pour saisir dans quelle zone se trouve la bullet.
- Lors de l'appel de OnChangeZone, tester si la newZone correspond à celle de la Bullet, et si oui, activer son déplacement.

11 - Physic materials

Il est possible de modifier comment un objet réagit physiquement avec les autres. Pour ceux-ci vous pouvez créer un Physic Material, que vous pouvez ensuite glisser dans le collider de l'objet. Vous pouvez y définir différents niveaux de friction et de rebond.