Date : 08/10/2024

Séance 5 – Cycle de vie d'un objet

Objectifs :

- Introduire les notions de cycle de vie d'un objet
- Mise en pratique des concepts décrits sur le moteur de jeu Unity

Concepts :

a) **Le cycle de vie d'un objet** (ou instance) d'une classe peut se diviser en trois étapes : sa création (ou instanciation), son utilisation et sa suppression (ou destruction). Le paradigme de programmation orientée objet définit des méthodes pour gérer les étapes des construction et destruction des objets, respectivement, les méthodes *constructeur* et *destructeur*.

Constructeur : cette méthode sert à paramétrer l'objet d'une classe avant son utilisation. Il est appelé une seule fois lors de l'instanciation de l'objet. Une classe peut avoir plusieurs constructeurs. Le constructeur par défaut ne prend pas d'arguments. Cependant, nous pouvons déclarer autant des constructeurs que nous en avons besoin, tant que chaque constructeur prend comme argument un ensemble de paramètres uniques par rapport aux autres constructeurs de la même classe.

Exemple de constructeur :

```
public class CoffreVoiture
{
    private float capacite;
//définition du constructeur par défaut
    public CoffreVoiture() {
        Debug.Log("Constructeur appelé");
        capacite = 0;
    }
//définition du constructeur alternatif
    public CoffreVoiture(float capaciteCoffre)
    {
        capacite = capaciteCoffre;
        Debug.Log("Constructeur appelé avec : "+capaciteCoffre+" litres");
    }
}
```

Déclaration d'objet avec appel au constructeur par défaut :

```
CoffreVoiture CoffreVoiture = new CoffreVoiture() ;
```

Déclaration d'objet avec appel au constructeur alternatif :

```
CoffreVoiture CoffreVoitureVoiture = new CoffreVoiture(5f) ;
```

Néanmoins, lorsque nous écrivons des scripts comportementaux Unity (MonoBehaviour), nous devons adopter les méthodes **« Awake »** et/ou **« Start »** pour la paramétrisation d'un objet **à la place** du **constructeur**. Ces méthodes seront aussi appelées automatiquement quand un objet sera instancié et **une seule fois** chacune dans leur cycle de vie.

La méthode « **Awake** » est couramment utilisée pour établir des références internes à un objet, comme les valeurs des attributs et des liens vers des objets extérieurs à la classe.

La méthode **« Start »** est lancée après la méthode « **Awake** » et juste avant l'exécution de la première « frame » du jeu. Elle sera donc utile pour le paramétrage de votre objet aussi. En revanche, la méthode « **Start** » est exécutée seulement si l'attribut booléen « *enabled* » du script comportemental vaut « *true* ». Notons que les méthodes « **Awake** » de tous les objets de la scène seront lancées avant que la première méthode « **Start** » ne soit appelée pour un objet.

À titre d'exemple d'utilisation des méthodes « **Awake** » et « **Start** », nous pouvons étudier le cas des jeux de tir. Dans ce contexte-là, la méthode « **Awake** » pourra être utilisée pour ajouter le joueur au jeu et approvisionner son inventaire, tant que la méthode « **Start** » sera utilisée pour lui accorder la permission de tirer sur ses ennemis.

Destructeur : cette méthode, aussi connu sous le nom de « finaliser », nous permet de faire les derniers traitements avant que l'objet ne soit détruit. Elle est lancée automatiquement par le « Coffre Voiture » de jeu lorsqu'un objet est signalé pour la destruction. Elle doit être utilisée essentiellement pour faire les derniers nettoyages relatifs à l'objet détruit, comme par exemple, signaler la destruction de ses sous-objets. Le destructeur d'une classe porte le même nom que la classe, mais avec le préfix « ~ ».

Exemple de déclaration d'un destructeur.

```
public class CoffreVoiture
~CoffreVoiture(){ //définition du destructeur
        Debug.Log("Destructeur appelé");
    }
}
```

De la même façon que pour le constructeur, un script « **MonoBehaviour** » doit utiliser une méthode alternative pour la destruction de l'objet : la méthode « **OnDestroy**() ». L'exemple ci-dessous montre une version alternative de la classe « CoffreVoiture » qui utilise les méthodes proposées par « MonoBehaviour » pour initialiser et détruire l'objet.

```
public class CoffreVoitureMB : MonoBehaviour {
    void Awake(){ }//constructeur – partie 1
    void Start(){ }//constructeur – partie 2
    void Update() { }
    void OnDestroy(){}//destructeur
}
```

Comme dans le framework .NET, le moteur de jeu Unity utilise une entité appelée « Garbage Collector » (GB) pour gérer l'allocation et la libération de mémoire de votre application. Chaque fois que vous créez un objet, la CLR alloue de la mémoire pour l'objet. Toutefois, la quantité de mémoire à notre disposition n'est pas illimitée. Le GB est donc exécuté périodiquement pour libérer de la mémoire allouée qui n'est plus utilisée. Une mémoire est considérée prête à être récupérée (ou recyclée) lorsqu'aucune variable ne fait plus référence à elle.

Nous pouvons utiliser la méthode « Destroy(objet) » pour demander explicitement la destruction des objets de types proposés par la bibliothèque Unity, tels que « Object », « GameObject » et « MonoBehavior », entre autres.

Exemple de destruction des objets - Type GameObject et Object

```
CoffreVoitureMB coffreVoitureMB1 = new CoffreVoiture();//création
Destroy(coffreVoitureMB1);//destruction
```

Néanmoins, la méthode « Destroy » ne convient pas à des objets créés à partir de classes standard (qui ne suivent pas le modèle « MonoBehaviour »). De plus, nous n'avons pas à cet instant une méthode pour demander explicitement la destruction de ces objets.

Les bonnes pratiques nous conseillent d'affecter la valeur « null » à un attribut (ou à une variable) qui fait référence à un objet dès qu'elle n'utilise plus cet objet. Une fois qu'aucun attribut (ou variable) ne fait référence à l'objet, le GB se chargera automatiquement de lancer sa destruction.

Exemple de destruction - Classe standard

```
CoffreVoiture = null;
```

Notons que c'est l'entité GB qui détermine le moment propice pour lancer une opération de collecte et de nettoyage de mémoire et que cette action peut avoir des conséquences sur la performance de votre jeu. Nous verrons plus tard des techniques utilisées pour bien gérer l'utilisation de mémoire dans un jeu.

Exercice 1 :

Pour les exercices suivants, créez un projet de jeu 3D sur Unity. Ensuite, ajoutez un objet de type « GameObject Empty » à votre scène avec un script comportemental nommé « Gestionnaire Jeu ».

a) **Constructeurs et destructeurs standard** :

1. Créez une nouvelle classe standard, nommé « Coffre Voiture », avec un constructeur par défaut et un constructeur alternatif. Ces constructeurs doivent afficher les messages suivants : « Le coffre de la voiture a été créé » et « Coffre de la voiture a été créé avec l'argument suivant : X ».
2. Créez des instances de cette classe dans le script « Gestionnaire Jeu » pour pratiquer l'utilisation des constructeurs par défaut et alternatif. Ces instances doivent être codées comme des attributs du script.
3. Ajoutez une méthode de type destructeur à la classe « Coffre Voiture » avec le message suivant : « le coffre de la voiture sera maintenant détruit! ».
4. Faites en sorte que les instances que vous avez créées de cette classe soient détruites lorsqu'on appuie sur une touche du clavier.

```
/*Le code ci-dessous vous permet de tester lorsque la touche « D » a été
relâchée après avoir été pressée.*/
Input.GetKeyUp(KeyCode.D)
```

Exercice 2 :

Reprenons maintenant notre projet Unity de la séance précédente avec une voiture.

b) **Constructeur pour script** :

1. Ajoutez un nouveau script « *Cycle Vie.cs* » à la voiture de votre projet de la séance précédente pour gérer sa construction et destruction.

2. Ajoutez les messages de texte suivants aux méthodes équivalentes au constructeur sur les scripts Unity : « *La voiture se réveille* » et « *La voiture finit son paramétrage juste avant son utilisation* ».
3. Sélectionnez votre voiture et décochez la case à côte du nom de ce script sur « *Inspector* ». Cette action désactivera le lancement de votre script vis-à-vis du moteur de jeu Unity.
4. Lancez votre jeu et vérifiez les messages de texte affichés.
5. Maintenant, arrêtez votre jeu, activez le script et relancez le jeu.

Questions :

- Avez-vous constaté des différences entre les résultats des instructions 4) et 5) ?
- Pouvez-vous faire appel aux méthodes Awake et/ou Start avec un argument ?

c) **Destructeur pour script :** ajoutez la méthode équivalente au destructeur au script « CycleVie.cs » avec le message : « *La voiture est en voie de destruction* ». De façon similaire à l' « *Exercice 1* **»**, ajoutez le code source nécessaire pour déclencher la destruction de la voiture à partir d'une touche du clavier. Faites attention que cette touche soit différente de celle utilisée par l'exercice précédent.

d) **Liaison entre scripts** :

1. Intégrez la classe « **Coffre Voiture** » au script « CycleVie.cs » sous la forme d'un attribut et initialisez-le dans la méthode appropriée.
2. Lancez votre jeu et déclenchez la destruction de l'objet « voiture ». Qu'observez-vous ?

Exercice 3 :

a) Téléchargez le prototype de jeu 2 proposé par Unity : Ressource Unity

b) Créez un nouveau projet de type « 3D » sur Unity et importez le « prototype 2 » dans votre projet.

Vous pouvez supprimer la scène rajoutée par défaut à votre projet afin de garder seulement la scène incluse dans le prototype.

c) Ajoutez un objet de type « Human », trois de type « Animals » et 1 objet de type « Food » dans votre scène.

Figure 1. Prototype 2 – Source : Unity Learn 2023

Utiliser la fenêtre « Inspector » afin de retourner de 180º les objets que représentent des animaux.

d) Créez un script « Contrôle Joueur » pour contrôler votre personnage et ajoutez à ce script la possibilité de déplacer horizontalement votre personnage (humain) dans la scène.

Vous pouvez utiliser les méthodes « GetAxis » ou « Translate » pour réaliser cette action.

e) Faites le nécessaire afin que le personnage ne puisse pas se déplacer au-delà du terrain de du jeu.

Parmi les solutions possibles, vous pouvez vérifier que la nouvelle position calculée pour le personnage est valide, et la modifier dans le cas contraire.

f) Créez un script qui déplace un objet vers l'avant (nommé « déplacement ») et rajoutez-le à l'objet de type « food ». Ce type d'objet jouera le rôle de projectile dans le projet.

Vous pouvez utiliser la méthode « Translate » et la direction « vector3.foward » pour accomplir cette action.

g) Créez un nouveau modèle d'objet (« prefab ») à partir de votre projectile (version avec le script) et enregistrez-le dans un dossier nommé « Prefabs »à l'intérieur du dossier « Assets ».

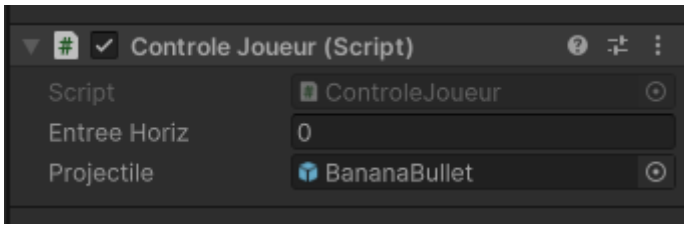
h) Rajoutez au script « Contrôle Joueur » la possibilité de tirer un projectile en utilisant la barre d'espace.

Pseudo-algorithme :

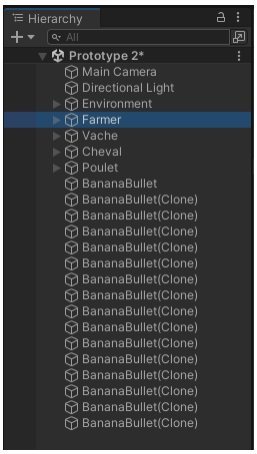
1. Utilisez la méthode « Input.GetKeyDown » afin de capturer l'événement d'appuyer sur la touche « espace ».
2. Créez un attribut de type « GameObject » dans votre script afin de définir le « prefab » à utiliser en tant que « projectile ».
3. Utilisez la méthode « Instantiate » afin de créer dynamiquement une instance de l'objet à tirer en utilisant l'objet stocké dans l'attribut « projectile ».

```
Instantiate(projectile,transform.position, projectile.transform.rotation);
```

4. Affectez le modèle d'objet crée, ici « BananaBullet », à l'attribut « projectile » en utilisant l'onglet « Inspector » d'Unity.



Vous pouvez constater qu'une fois tirés, les projectiles restent dans votre scène, même s'ils ne sont plus affichés.



i) Créez un script que détruit l'objet « GameObject » auquel il est attaché à partir du moment que l'objet sort de la scène. Rajoutez ce script au « modèle » de projectile créé précédemment (ou créez un nouveau modèle).

Vous pouvez utiliser la méthode « Destroy » vue au début de la séance pour détruire les projectiles.

j) Utilisez la procédure apprise pendant les exercices précédents afin de déplacer automatiquement les animaux du haut vers le bas de la scène (questions f à i de l'exercice 3). Faites le nécessaire afin que les objets de type « animal » soient aussi détruits lorsqu'ils dépassent les bords du terrain.

Considérations finales

Une partie des exercices de ce TD a été adaptée de l'unité 2 du Parcours Unity Learn - Junior Programmer : Create with Code. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/destructors
- https://answers.unity.com/questions/513439/when-should-i-use-constructors-vs-using-awake-or-s.html
- https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html
- https://docs.microsoft.com/fr-fr/dotnet/standard/garbage-collection/
- https://learn.unity.com/tutorial/lesson-2-3-random-animal-stampede?uv=2021.3&missionId=5f71fe63edbc2a00200e9de0&pathwayId=5f7e17e1edbc2a5ec21a20af&contentId=5f7229b2edbc2a001f834db7&projectId=5cdcc312edbc2a24a41671e6#